Applying MPEG DASH in the browser for non-traditional video streaming scenarios

by

Geoffrey Beertens

Promotor: Prof. Dr. P. QUAX
Mentor’s: Dr. Maarten Wijnants & Gustavo Rovelo
2014–2015
Abstract

Video streaming has become increasingly popular in recent years. To be able to have interoperability in this field a standard is created, called MPEG-DASH. In this thesis, we conduct a case study to take a look at the possibilities of creating advanced video streaming use cases using MPEG-DASH in Web browser environments. By implementing the use cases in a Web browser environment, it will be able to run on multiple platforms. In these use cases the goal is to have advanced video streaming techniques while at the same time being bandwidth efficient. We studied two advanced video streaming techniques in the context of DASH. The first technique is based on non-linear video to create some kind of virtual tour. Here we created a proof-of-concept application that is able to seamlessly switch between multiple sequences of video footage in the non-linear video. Secondly, we looked at a technique to enlarge the amount of data the user sees when watching a video. This technique is especially useful when watching an Omni-Directional Video where only a part of the video is visible at any given time. In this technique we use multiple screens to show a bigger part of the video at all times. When using this technique in an ODV context, we show more video footage of the camera’s surroundings while maintaining the same level of detail. However, sending high resolution videos over a network with limited bandwidth capabilities can be troublesome. To counteract this problem we created a solution to only send a part of the video in high quality while sending the rest in a lower quality. We generally prioritize a part of the video that has the user’s focus. We developed two techniques to send parts of the video in different qualities. First, we created a tiling approach to be able to assign priority to some tiles. Secondly, we are able to segment the video in different layers of detail. Objects that have the focus of users are generally put in the top layer. Possible less-important background data can be put in the lowest layer.
Acknowledgments

First and foremost, I would like to thank my promotor Peter Quax, my co-promotor Wim Lamotte, Maarten Wijnants and Gustavo Rvelo for their guidance and feedback on my thesis. I also thank Pieter Vincken and Geoffrey Martens for their suggestions in context of a similar master thesis subject and my fellow students for their general support. I would like to thank every participant in the user study for their valuable feedback. Lastly, I thank my friends and family for their endless support during the course of this thesis.
# Contents

1 Introduction 9

2 History of video streaming 11

I Literature study 14

3 MPEG DASH 15
   3.1 Overview 15
   3.2 Media Presentation Description 17
   3.3 Segments 20
   3.4 Video/audio codecs 22
   3.5 Profiles 22
   3.6 Adaptive streaming 23
   3.7 Quality of Experience 26
   3.8 DASH implementations 27
   3.9 DASH tools 28

4 Videos in webbrowser environments 29
   4.1 HTML5 Video 29
   4.2 HTML5 Canvas 31
   4.3 Media Source Extensions 32
   4.4 WebGL 33
   4.5 ODV Viewer 34

II Use cases 37

5 MPEG-DASH Implementation Framework 38
   5.1 Introduction 38
   5.2 Backend software architecture 39
   5.3 Frontend libraries 42
   5.4 Configuration options 43
   5.5 External libraries 44

6 Path-Based Non-linear Video 46
   6.1 Introduction 46
   6.2 Related Work 48
   6.3 Approach 50
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Mobile Internet traffic growth in the period 2013-2018 [16]</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Global Internet traffic growth in the period 2011-2016 [67]</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Scope of DASH [36]</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Data model of an MPD</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Profile scope of DASH</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>A video frame showing the 360 degree field of view of an ODV.</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Screenshots of the ODV Viewer while using a 3D spherical renderer.</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Screenshots of the ODV Viewer while using a 2D planar renderer.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>The structure of the DASH implementation framework</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>The modular software architecture of in the backend</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>The communication library abstracts the communication between</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>a web-based frontend on the one hand and a backend and gesture-tracking server on the other hand.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>The visualization library structure</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>An image of a non-linear video for the Smithsonian national museum</td>
<td>47</td>
</tr>
<tr>
<td>6.2</td>
<td>Example use of the round robin solution by [45]. The terms used in this example are different from the terms used in this thesis:</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>chunks are segments, segments are sequences and branch points are intersections in the non-linear video.</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>The graph created from the media experience example in Listing 6.1</td>
<td>52</td>
</tr>
<tr>
<td>6.4</td>
<td>A screenshot of the HTML5 frontend for the choose your own path use case</td>
<td>54</td>
</tr>
<tr>
<td>6.5</td>
<td>A screenshot of the QT frontend for the choose your own path use case</td>
<td>55</td>
</tr>
<tr>
<td>6.6</td>
<td>Graph representation in the frontend with different statuses for some nodes</td>
<td>56</td>
</tr>
<tr>
<td>6.7</td>
<td>Median for user test: transition time. The chart displays the median of the rating for visibility and acceptance of the transition delay. The transition delay varied from 50 ms to 250 ms. It also displays the same results for our solution.</td>
<td>59</td>
</tr>
<tr>
<td>6.8</td>
<td>Test1: a single low quality representation with the always lowest quality and parametric precaching</td>
<td>63</td>
</tr>
<tr>
<td>6.9</td>
<td>Test2: a single low quality representation with always lowest quality and parametric precaching</td>
<td>64</td>
</tr>
</tbody>
</table>
6.10 Test3: a single medium quality representation with always lowest quality and precaching at the beginning.

6.11 Test4: a nine representations media experience with always highest quality and precaching at the beginning.

6.12 Test5: a nine representations media experience with always highest quality and parametric precaching.

6.13 Test6: a nine representations media experience with best fit and precaching at the beginning at 18 Mbps.

6.14 Test7: a nine representations media experience with best fit and parametric precaching at 18 Mbps.

6.15 Test8: a nine representations media experience with best fit and parametric precaching at 9 Mbps.

6.16 Test9: a nine representations media experience with four children with best fit and parametric precaching at 18 Mbps.

7.1 One frame of an ODV where we highlight the part of Figure 7.2. This could be the viewport of an ODV Viewer or a separate video file.

7.2 The same information as highlighted by the box in Figure 7.1.

7.3 The division of an ODV into four tiles: top left, top right, bottom left and bottom right. As a visual guideline we drew the tile boundaries on top of the video. In a tiled video application, these guidelines are not drawn.

7.4 The process to create a tiled video, where we have a fixed number of tiles and each tile can have different resolutions.

7.5 Evaluated use cases for tiled HTTP adaptive streaming (TAS).

7.6 Our tiling approach in the case of four tiles. Each tile has the same size and can have different qualities.

7.7 An entire ODV frame where we highlight the viewport of Figure 7.8. The viewport is located on the right and left boundaries of these ODV frame.

7.8 An ODV viewport that horizontally "wraps around" the edges of the video footage.

7.9 An image explaining the viewport to viewers relation where we will need the circular tests in Listing 7.3. We clearly see that the end of the viewport can fall outside of the canvas. The viewport will wrap around the edges of the canvas.

7.10 A panning process in the high quality viewport use case.

7.11 Baseline for the following tests. Consumed bandwidth for a high quality stream of bitrate 6000 Kbps.

7.12 Legend used for the 16 tiles. These colors are used for coloring the stacked portions in the following tests.

7.13 Consumed bandwidth for all tiles in high quality.

7.14 Consumed bandwidth for one tile in high quality.

7.15 Consumed bandwidth for two tiles in high quality.

7.16 Consumed bandwidth for four tiles in high quality.

7.17 Consumed bandwidth for eight tiles in high quality.
7.18 A chart displaying a comparison between an overall lower quality video and a high quality viewport video both usable under limited bandwidth conditions. The comparison is made using four values: median, mode, minimum and maximum. 87

7.19 A chart displaying the median of the participant’s ratings for different configuration options of the high resolution viewport technique. Each configuration is named by the three important parameters: switching time, quality outside of the viewport and panning distance. 89

7.20 A chart displaying the median of the participant’s ratings for configurations with low quality outside the viewport and with varying switching time and panning distance. 90

8.1 A frame from respectively the background and foreground video created by segmentation of the ODV using the rotoscoping tool in Adobe After Effects. 95

8.2 A frame of a multi-layered video in which the data for the actor is located in the foreground layer. 97

8.3 A multi-layered video with the red car as priority object. The car is sent in a higher quality than the background. The green chroma key pixels are filtered out of the foreground layer. 98

8.4 An example of a consumed bandwidth chart for a high resolution video stream. Streaming this video in limited bandwidth conditions can be troublesome. This non-segmented video is used to create two separate segmentations for this test. 99

8.5 Two examples of multi-layered video streams. The foreground layer contains the actor, who is always visible. The background and foreground layers are displayed in a stacked graph and are respectively represented by ind01 and ind02. In the two examples we varied the quality of the background. The left example has a lower background quality. 100

8.6 Two examples of multi-layered video streams. The foreground layer contains the red car, which is visible for a short period of the video and covers a small portion of the video the rest of the time. The background and foreground layers are displayed in a stacked graph and are respectively represented by ind01 and ind02. In the two examples we varied the quality of the background. The left example has a lower background quality. 100

9.1 An example of a CAVE setup. This CAVE setup is located in the National Center for Supercomputing Applications at University of Illinois at Urbana-Champaign [19]. 105

9.2 A example of how three screens can be positioned within the multi-screen setup. 105

9.3 A diagram showing how the connection between multiple components takes place in the implementation framework. Only one client, the master, connects to the backend. Every client connects to the middleware. Because this middleware is a part of the gesture-tracking server, we will be able to interact with the setup using gestures. 109
9.4 Setup options in a web-based client for the multi-screen setup. The client’s role and spatial location can be specified with these options.

9.5 An example of a multi-screen setup used at EDM. The left, middle and right screens respectively have the following roles: master, slave and middleware.
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Browsers and their support for MP4, WebM and Ogg video content on desktop platforms</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>A list of some HTML5 video methods, properties and events and their explanation</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Results of 10 runs of the experiments for each stream in the dataset. We did the test for both scenarios. The results are displayed in milliseconds</td>
<td>58</td>
</tr>
<tr>
<td>6.2</td>
<td>The post-hoc comparison of all transition time combinations. A * is used when a significant effect was found between two transition times. When no significant effect was found, a – is used. We only use the lower part of the table to avoid visual clutter and to not include duplicates</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Result of performance test in low, medium and high quality. The wo and w stand respectively for active stream without or with precaching. All the results are displayed in percentages of time spent in the function compared to the total time of the test</td>
<td>62</td>
</tr>
<tr>
<td>6.4</td>
<td>All configuration permutations tested during this test. We use abbreviations to conserve space inside the table. B stands for precaching at the beginning. P stands for parameteric precaching. AL, AH and BF respectively stand for always lowest quality adaptation, always highest quality adaptation and best fit quality adaptation. The stream quality can be low, med (medium) or varying (/) thus having multiple possible qualities. Available bandwidth is expressed in Mbps</td>
<td>64</td>
</tr>
<tr>
<td>9.1</td>
<td>Results of four tests of the multi-screen approach. In the first two tests we used a master-slave approach. In the last two we used a stand alone client, thus did not send any data to the middleware. All the results are displayed in milliseconds (ms)</td>
<td>113</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Popularity of video streaming has increased a lot these past years and is still growing. Currently, smartphones and tablets have become more familiar to the general public. On these devices watching streamed videos is a popular occupation. Video traffic accounts for a large amount of the mobile data transfer. Users generally have high expectations concerning video streaming, even in mobile environments, such as high visual fidelity and fast start-up.

For many years, the browser is one of the applications used to display streamed videos. Before the HTML5 standard, browser specific implementations or third-party plug-ins had to be used to render videos on a webpage. Browser specific implementation lack interoperability between browsers. Plug-ins have two major disadvantages: video rendering is done in a black box and were often not available for mobile platforms. The introduction of the HTML5 video tries to solve all the disadvantages that these browser specific implementations and third-party plug-ins have. With the HTML5 video, there is interoperability between multiple browsers and developers are able to access the videos during rendering.

The preferred method of streaming data over the Internet has shifted from the Real-Time Streaming Protocol (RTSP) to the HTTP protocol. RTSP is no longer used because it had some big disadvantages: the protocol is often blocked by firewalls, it is often not supported by Content Delivery Networks (CDNs) and the server has to maintain a state for each client. These disadvantages are solved by using HTTP as a transport protocol. Therefore, HTTP streaming is adopted by the major commercial streaming platforms such as Apple’s HTTP Live Streaming, Microsoft’s Smooth Streaming, and Adobe’s HTTP Dynamic Streaming. These platforms all provide adaptive streaming functionality but different kind of implementations. To stimulate the growth of video streaming, interoperability has to be increased. To achieve this a video streaming standard, called MPEG-DASH is created.

Dynamic Adaptive Streaming over HTTP (DASH) is a standard created by the MPEG. This standard does not define client- or server-side functionality but rather focuses on the used data formats. The DASH formats can be used to stream data from any standard HTTP server to HTTP clients. There are two formats: the Media Presentation Description (MPD) and segment formats. Because MPEG-DASH is created as a new standard for adaptive streaming it enjoys a lot of features. Some of these features are adopted from the commercial
streaming platforms and others are expected to be useful or mandatory in the near future. The MPEG-DASH standard does not define what transfer protocol has to be used, regardless of this in practice HTTP is mostly employed.

In this thesis we focus on video streaming in Web browser environments. The main research question is: “Does the MPEG-DASH standard allow for the deployment of advanced and bandwidth efficient streaming use cases in Web browser environments?” This question will be answered by a case study, where we study multiple advanced and bandwidth efficient DASH related use cases. Each use case addresses unique functionality to show the possibilities of MPEG-DASH in a Web browser environment. We want to study how MPEG-DASH can be used to create advanced use cases. It is important that these use cases are bandwidth efficient to be able to adopt these techniques when bandwidth is limited, for example when using mobile Internet.

Concretely, four use cases will be studied: “Path-Based Non-linear Video”, “Tiled ODV distribution with viewport resolution optimizations”, “Multi-layer” and “Multi-screen”. These use cases address advanced video streaming scenarios and bandwidth problems with streaming high resolution videos. In “Path-Based Non-linear Video” we study how non-linear video can be streamed with DASH. To showcase this functionality, we create a proof-of-concept demo. In this thesis we often use high resolution videos with a 360 degree field of view, these videos are called Omni-Directional Videos (ODV). Because streaming high resolution videos can be a problem in limited bandwidth conditions we study two techniques to lower the consumed bandwidth. We try to send a possible less-important part of the video in a lower quality to be able to enjoy the focus of the video in high quality. In “Tiled ODV distribution with viewport resolution optimizations” we use tiles to be able to select the quality of each region. In “Multi-layer” multiple layers are used to differentiate between quality levels. In “Multi-screen” we study a technique to enlarge the amount of video data a user can view by using multiple screens.

We start this thesis with the history of video streaming in Chapter 2. Afterwards, we give a detailed explanation of MPEG-DASH in Chapter 3. In Chapter 4 we look at available browser techniques to stream videos. The remainder of the thesis will consist out of use cases. Chapter 5 describes the framework used for each use case. Chapters 6, 7, 8 and 9 each describe a unique use case and finally we will come to a conclusion in Chapter 10.
Chapter 2

History of video streaming [70] [69]

With the recent rise in popularity of smartphones and tablets, Internet access on mobile devices has drastically increased these last few years. A large amount of mobile data transfer can be attributed to video traffic. In Figure 2.1 we can see that if the growth would continue at the current rate, 69% of all mobile data will be video traffic in the year 2018. Even though users are increasingly watching videos on a mobile device, they still expect a high quality video experience. Factors that influence the video experience are: high visual fidelity, fast start-up, fast reactivity to user interaction, trick mode support, et cetera.

![Figure 2.1: Mobile Internet traffic growth in the period 2013-2018][1](#)

High data volumes caused by online video distribution is not only a trend in mobile Internet access but can be extended to the global Internet traffic. In Figure 2.2 we can see that video data accounts today for half of all Internet traffic. Even though these numbers seem high, video streaming is still in its infancy. It has the potential to account for an even larger part of the Internet traffic.
traffic. The reason for this is the lack of interoperability between various servers and clients. At the moment every commercial platform uses proprietary manifest formats, content formats and streaming protocols, this way giving rise to closed solutions without interoperability support.

![Figure 2.2: Global Internet traffic growth in the period 2011-2016](image)

There are two main methods for streaming video data over the Internet: traditional streaming which involves the Real-Time Streaming Protocol (RTSP), and streaming over HTTP. In the 1990s RTSP was created with a focus on in-time delivery and the ability to handle large amounts of data. RTSP is a stateful protocol, meaning the server has to maintain a state for every client connection. RTSP can deliver small packets with low overhead and works well in managed IP networks. Unfortunately, RTSP has some big disadvantages: the protocol is often blocked by firewalls, maintaining a state for every client does not scale well and RTSP is often not supported by Content Delivery Networks (CDNs). With the growth of the Internet, it has become possible to transfer larger packets of data using HTTP. HTTP can be used to distribute video content. We will call transferring larger packets of data over HTTP, HTTP streaming. HTTP is in contrast to RTSP a stateless protocol, every client has to manage its own state. HTTP streaming has multiple advantages: it is firewall friendly, it is supported by CDNs and because it is stateless it scales really well. For all these reasons, HTTP streaming became the method of choice for commercial streaming platforms such as Apple’s HTTP Live Streaming [37], Microsoft’s Smooth Streaming [51], and Adobe’s HTTP Dynamic Streaming [3].

To stimulate the growth of video streaming, interoperability needs to increase. To achieve this, a standard for HTTP streaming of multimedia content needs to be made. This will allow for a standard-compliant client to stream content from any standard-based server. MPEG carried out the task to create a standard in collaboration with other standard groups such as Third Generation Partnership Project (3GPP). The result is called MPEG Dynamic Adaptive Streaming over HTTP (DASH) [1]. DASH takes advantage of HTTP streaming
and tries to fulfill the high expectation of users regarding video experience. The choice was made to create a new standard rather than “promote” one of the commercial streaming platforms to a standard [44]. By creating a new standard, it was feasible to adopt features from all three platforms and anticipate new features as well as to provide built-in support for them.

When no standard way was specified to include a video in a web page, developers had two options [18]. The first option was to use browser-specific implementations. A second option was to resort to third-party plug-ins such as VLC plug-in, Apple QuickTime or Adobe Flash Player. These plug-ins can be included in a web page with the use of a <object> tag. For a long time developers had to rely on such plug-ins to do the video rendering. The first disadvantage of the plug-in-based approach is that it transforms rendering video into a black box for the web browser. This means that styling with CSS and masking and filtering with SVG is not an option. We can not change the look-and-feel of the video player because there is no standard way to interact with the black box through Javascript. The next disadvantage is that these plug-ins are rarely available on mobile devices, only on desktop-like platforms. With the recent popularity of mobile devices and video streaming, we surely want a solution that is able to provide a good video experience for a mobile user. Such a solution is created in the HTML5 standard, the HTML5 video.

Video streaming in web browser environments has evolved heavily in recent years. It started with plug-in-based rendering and transport via RTSP. From that point forward, the video player evolved to a HTML5 video and the transport protocol to HTTP. The use of HTTP enabled streaming platforms such as Apple’s HTTP Live Streaming, Microsoft’s Smooth Streaming, and Adobe’s HTTP Dynamic Streaming to adaptively stream video content depending on environmental conditions like, for example, the prevailing bandwidth available between the client and the video server. Because there was a lack of interoperability in video streaming, a standard was created called MPEG DASH.
Part I

Literature study
Chapter 3

MPEG DASH

3.1 Overview

Dynamic Adaptive Streaming over HTTP (DASH) is a standard that focuses on data formats to provide a DASH Media Presentation rather than on client- or server-side procedures. DASH specifies XML and binary formats that provide the ability to deliver media content from standard HTTP servers to HTTP clients. However, DASH does not define what delivery protocol must be used for the media content delivery. DASH intends to support a media-streaming model where the client has full control over the delivery of media content. A DASH client may use HTTP GET or partial GET requests to receive content from standard HTTP servers, even if the HTTP servers have no DASH-specific capabilities. In Figure 3.1 we can see the scope of DASH as described above.

The DASH standard specifies two formats: the Media Presentation Description (MPD) and the segment formats. An MPD describes a manifest of the available media content, which is expressed in terms of so-called segments. To provide a smooth playback, the MPD contains timing information for a DASH client to use for playback purposes. The segment formats specify the syntax of

Figure 3.1: Scope of DASH.

The DASH standard specifies two formats: the Media Presentation Description (MPD) and the segment formats. An MPD describes a manifest of the available media content, which is expressed in terms of so-called segments. To provide a smooth playback, the MPD contains timing information for a DASH client to use for playback purposes. The segment formats specify the syntax of
the body of a HTTP GET or partial GET response. In most cases segments contain encoded media data and metadata that are encapsulated in common media formats.

MPEG-DASH also has some additional features which derive from the way DASH is defined:

- Switching and selectable streams. DASH lets the client decide what representation of the data it wants to use both initially and during playback.
- Trick mode support such as seeking, fast forward or rewind.
- Ad insertion. Advertisements can be inserted in between segments.
- Segments with variable durations.
- Multiple base URLs. The same content can be hosted at multiple locations. This maps well to the use of CDNs.
- Clock-drift control for live sessions. The server’s UTC time can be included with each segment to enable the client to control its clock drift.
- Scalable Video Coding (SVC) and Multiview Video Coding (MVC) support.
- Content protection and transport security.
- Quality metrics for reporting the session experience.
- ...

The HTTP delivery protocol is not part of the DASH standard but is widely used in DASH contexts as a transport protocol to provide content to DASH clients. Using HTTP has numerous advantages, some of the most important ones being: it avoids NAT and firewall issues, it provides reliable transfer due to the underlying TCP/IP protocol, it provides the ability to use standard HTTP caches or CDNs, clients maintain their own state, and it provides client the ability to request data on-the-fly. Because of this, we will consider HTTP as the primary delivery technology for DASH data in the remainder of this thesis.

To play DASH media content, the DASH client first has to obtain the MPD pertaining to the desired content. This MPD can be obtained in various ways, for example through HTTP via an URL. Next, the client has to parse the MPD to learn about the media content’s timing, media types, resolution, bandwidth, encoded alternatives, et cetera. Using this information, the client can decide which segments it needs and can start fetching them via HTTP GET requests. The client will have to take network conditions and buffer availability into account to provide a smooth playback. For example, the client has to monitor the available bandwidth and adapt to bandwidth fluctuations by switching to an alternatively encoded version of the content with possibly a lower bitrate.
3.2 Media Presentation Description

In the Section 3.1 we already mentioned that a Media Presentation Description (MPD) describes a manifest of the available media content. The MPD is defined in such a way that it enables the client to control every aspect of the streaming process. In this section we will take a closer look at the type of information a client can expect in such an MPD.

An MPD describes a Media Presentation. A Media Presentation is a collection of encoded and deliverable versions of media content and their metadata. The media content is composed of contiguous temporal periods. Each such period consists of one or more media content components, for example a video component and an English audio component. Each media content component has a type, for example video or audio. Every media content component may have various encoded versions of the media content, we call these versions media streams. A media stream will also provide metadata about the media content: the media content period, the media content component it is part of, codec parameters, encoding bitrate and et cetera.

![Data model of an MPD](image.png)

Figure 3.2: Data model of an MPD.

An MPD is an XML document with a set of predefined elements. In Figure 3.2 we can see an abstract representation of the data model of an MPD. We will now explain every aspect of this data model in detail. The top level of an MPD consists of a sequence of periods in time. Such a period typically represents a media content period where the set of encoded versions of the media content does not change.

A period is divided into adaptation sets, which represent a set of interchangeable encoded versions of media content components. To clarify this, a likely scenario would involve distinct adaptation sets for the main video, the main audio and perhaps captions.

An adaptation set exists of a series of representations. A representation contains a media stream, which is a single encoded version of a media content component. To be able to render one media content component, it is sufficient to render only one representation of the adaptation set. Within one adaptation set it is customary to switch between representations when factors of the playback change, for instance the network conditions. When a representation contains a media stream that uses a codec that the client does not support, it can be
ignored for the rest of the playback.

A representation is divided into **segments**. These segments are the media stream chunks in temporal sequence. Each segment has an URI that can be accessed by the client via a HTTP GET or partial GET request. A segment is the largest unit that can be retrieved by a single HTTP request.

A key feature of DASH is the existence of a common timeline between differently encoded versions of media content components. This common timeline is called the Media Presentation timeline and is constructed using the presentation time of an access unit within the media content component. Segments contain timing information to make synchronization between components possible. At the same time, it provides the ability to seamlessly switch between representations.

We will now take a closer look at an MPD document. We will inspect this MPD at different granularities of detail. First, we will look at the most abstract level in Listing 3.1. In this listing we see the basic structure of the MPD as explained above. We can clearly see that the example MPD exists out of only a single period. The period exists out of two adaptation sets. Each adaptation set holds two representations. Each representation has multiple segments.

```xml
1 <MPD ...>
2   <Period ...>
3     <AdaptationSet ...>
4       <Representation ...>
5         <SegmentList ...>
6           <Initialization .../>
7           <SegmentURL .../>
8           <SegmentURL .../>
9           <SegmentURL .../>
10          ...
11       </SegmentList>
12     </Representation>
13     <Representation ...>
14       <SegmentList ...>
15           <Initialization .../>
16           <SegmentURL .../>
17           <SegmentURL .../>
18           <SegmentURL .../>
19          ...
20       </SegmentList>
21     </Representation>
22   </AdaptationSet>
23   <AdaptationSet ...>
24     <Representation ...>
25       <SegmentList ...>
26           <Initialization .../>
27           <SegmentURL .../>
28           <SegmentURL .../>
29           <SegmentURL .../>
30          ...
31      </SegmentList>
32     </Representation>
33     <Representation ...>
34       <SegmentList ...>
35           <Initialization .../>
36           <SegmentURL .../>
37           <SegmentURL .../>
38           <SegmentURL .../>
39          ...
```

18
Listing 3.1: Example MPD with only the high level elements visible

Listing 3.2 shows an example of an adaptation set where segments are aligned. The adaptation set has two important attributes: segmentAlignment and bitstreamSwitching. When segmentAlignment is set to true, segments of representations within this adaptation set are aligned. This means that there is no overlap between segments from different representations which do not have the same index. For example, segment 1 of representation X does not overlap with segment 2 of representation Y. BitstreamSwitching with a value of true means: all representation of the adaptation set have the same number of segments. A sequence of consecutive segments created from these representations will result in a “conforming segment sequence”, as defined in the standard.

Listing 3.2: Example of an adaptation set where segments are aligned

In Listing 3.3 an example is shown of an adaptation set where subsegments are used in its representations. This example also has two important attributes: subsegmentAlignment and subsegmentStartsWithSAP. SubsegmentAlignment in essence means the same thing as segmentAlignment but for subsegments. When subsegmentAlignment is set to true, each media segment has to be indexed. We will give more information about this indexing in the next section. SubsegmentStartsWithSAP with value 1 means that each subsegment starts with a Stream Access Point.

Listing 3.3: Example of an adaptation set where subsegments are used in its representations

In listings 3.4, 3.5 and 3.6 we see examples of how segments can be provided using different methods. We respectively see the use of a segmentList combined with SegmentURLs, a representation as a single segment and a segmentTemplate. All these representations have multiple attributes in common. We will only explain those that have a non-trivial explanation. We explain the following attributes: id, sar and bandwidth. The id is used to identify the representation. Within one period, the id is unique. The only case when an id is not unique, occurs when two representation have exactly the same functionality. Sar stands for sample aspect ratio of the video media component type. The two numbers are respectively the horizontal and vertical size. Bandwidth contains a value that represents a constant bitrate channel of bandwidth in bits per second (bps). If segments are provided with this bitrate, the stream will be able to play without interrupts. As we can see in these listings, representations can have segment information assigned using different methods. There are four different elements: BaseURL, SegmentBase, SegmentTemplate or SegmentList.
There are also two attributes that are contained in the MPD field which are important for the remainder of this thesis: mediaPresentationDuration and minBufferTime. The mediaPresentationDuration specifies the duration of the entire Media Presentation. For example, when only one representation is specified, the duration will be the length of the media described by the representation. The minBufferTime specifies the amount of time a client has to wait after receiving the first byte of the data.

For on-demand applications, the MPD is a static document describing the Media Presentation because all the segments are readily available server-side. However, when using live applications, the MPD has to be updated regularly to include information pertaining to the newly generated segments. In such an application, the MPD will be gradually built to reflect the changes over time.

We now know what kind of information is stored in the MPD, but how can a client use this information? A client will first have to obtain the MPD. Once he has this MPD, he will have to parse it using an XML parser. The result of this parsing will provide the client with all the information needed to start the playback.

### 3.3 Segments

In the Section 3.1, we learned that the segment formats specify the syntax of the body of a HTTP GET or partial GET response. This means that when we do a HTTP GET request for a segment from the MPD using its URL, we will
get a HTTP response with the entire body being a segment. In this section we will give a more broad description of segments, treating in more detail segment types and segment formats.

Four types of segments are defined:

- Initialization segments
- Media segments
- Index segments
- Bitstream switching segments

The initialization segments contain initialization information for accessing the representation. They do not hold any actual media data but instead contain information to initialize the DASH client’s decoder.

The media segments contain data from the media stream that is associated with the involved representation. In addition, it will contain a number of complete access units and at least one Stream Access Point (SAP). This SAP is a random access or switch-to point in the media stream where the decoder can only use data from that point forward. Media segments may contain multiple subsegments. Each subsegment will contain a number of complete access units. The division of a segment is described in a segment index, which provides timing information to map the subsegments to the correct timeline of the segment. A client can use this information to receive subsegments by doing partial HTTP GET requests. The indexing information can be located at the beginning of a segment or spread over multiple indexing boxes in the segment. Indexing information may also be provided in separate index segments. Different methods of spreading index information are possible, such as hierarchical, daisy chain and hybrid. This last spreading technique avoids adding a large box of indexing information at the beginning of the segment to prevent initial download delay.

The index segments are related to the media segments and contain indexing information. It can provide information for more than one media segment and are format specific.

The bitstream switching segments contain information for switching to a given representation. These kind of segments are also format specific. If the bitstream switching flag in the MPD is set to true no bitstream switching segments will be present.

There are two segment formats:

- Media segments based on the ISO Base Media File Format
- Media segments based on the MPEG-2 Transport Stream

Both ISO Base Media File Format and MPEG-2 Transport Stream are multimedia container formats, which specify the relation between data and meta data in the multimedia file. ISO Base Media File Format is a container for audio, video and text and specified in ISO/IEC 14496-12 [68]. ISO Base Media File Format is used as a basis for other media file formats, with the most popular formats being MP4 and 3GP. The MPEG-2 Transport Stream is a standard container for digital broadcasting and for transportation over unreliable media. It is specified in MPEG-2 Part 1 [31]. It is used as a basis for the Blu-ray Disc Audio-Video MPEG-2 Transport Stream (M2TS).
3.4 Video/audio codecs

One of the big advantages of MPEG-DASH is that it is codec agnostic [24]. This means that all codecs are supported by DASH as long as they can be described with the two segment formats mentioned in Section 3.3. This also means that new codec options are naturally supported. The burden is put on the encoder and player builders to support these codecs. When developing a DASH client, we have to determine which codecs we want to support and how the encoder generates segment encapsulations for this codec. Even though a lot of codec are supported, MP4 H264/AAC is mostly used in the video streaming industry [50]. In this thesis, we will always use MP4 files to create our DASH datasets, therefore being in line with the most used codec in the industry.

3.5 Profiles [69] [71] [1]

Profiles of DASH are created with the focus on interoperability. A profile can be seen as a set of restrictions for DASH clients that only need to implement the features required by that profile. These restrictions often apply to features of the MPD and to segment formats. However, it is also possible that a profile imposes a restriction on the content within a segment, such as on media content type, media format, codec, bitrate, segment duration, segment size and so on.

Profiles are described within an MPD by the attribute @profiles. Every profile will be identified by an URI, separated by commas in the @profiles field. These profiles can also be used to check if the MPD is valid. Every profile within the @profiles field sets a series of restrictions on the MPD, so the MPD has to conform with all of them.

The current DASH standard includes six profiles:

- ISO Base Media File Format Full profile
- ISO Base Media File Format Main profile
- ISO Base Media File Format On Demand profile
- ISO Base Media File Format Live profile
- MPEG-2 TS Main profile
- MPEG-2 TS Simple profile

It is still possible to define other profiles that set other restrictions on the MPD and segments. Such an external profile can be identified with an URL. This URL has to include a date to avoid problems with domain names changing ownership.

We can get a notion of how these profiles relate to each other in Figure 3.3. We immediately see that the ISO Base Media File Format Full profile encapsulates every other profile. To be compliant with the Full profile, a DASH client has to implement all DASH related features defined by the standard. The same interpretation goes for the ISO Base Media File Format Main profile. This profile contains the features of both the On Demand and Live Profile. We will now take a look at what relevance each profile has and what restrictions it introduces. The ISO Base Media File Format On Demand profile is...
intended to be able to support On Demand content. It supports large Video on Demand (VoD) libraries with minimal content management. It focuses on scalable and efficient use of HTTP servers and provides a simple way of seamless switching. The most important restrictions it puts on the MPD are: each representation is provided as a single segment, subsegments are aligned across representations within an adaptation set and subsegments begin with a Stream Access Point. The ISO Base Media File Format Live profile is created to support live encoding and low latency delivery of short segments. Each segment can be requested when it is available, this way eliminating the need to update the MPD before requesting new segments. The Live profile introduces a restriction on the segments so that these can be concatenated on their boundaries to avoid gaps or overlaps in the media data even when switching representations within an adaptation set. The profile also includes the functionality to provide an On Demand experience from previous live content by modifying the MPD type field to static. The MPEG-2 TS Main profile sets only two constraints on the media segment format for MPEG-2 TS: representations have to be multiplexed and segment formats have to be split on MPEG-2 TS packet boundaries. HLS content can be integrated within this profile. The MPEG-2 TS Simple profile can be used to allow a simple implementation of seamless switching. The Simple profile introduces more restrictions on encoding and multiplexing.

A media engine conform to ISO/IEC 13818-1 is able to play a bitstream by concatenating consecutive segments from any representation within the same adaptation set.

The focus of this profile is to be conform with so when we concatenate consecutive segments from any Representation within the same Adaptation Set we can play the bitstream.

3.6 Adaptive streaming

DASH stands for Dynamic Adaptive Streaming over HTTP, but what exactly is adaptive streaming? Adaptive streaming is on-the-fly adjustment of streaming, to accommodate for changing playback conditions. To give an example one of those changing playback conditions could be a drop in available bandwidth.

Figure 3.3: Profile scope of DASH.
when playing a video stream. In an adaptive streaming scenario the streaming
client should provide a solution to such reductions in available bandwidth. One
possibility would be to lower the video quality to reduce the impact the video
streaming has on the bandwidth consumption. Adaptive streaming not only re-
acts to network conditions but also to user generated events and other playback
factors. For example, a user can choose that he wants a better quality of video.
Adaptive streaming in general is meant as a way to provide a smooth playback
for the user.

For some users it may be sufficient to only provide one media stream without
adapting to playback conditions. However, for most it would not. There are way
too many factors influencing the media playback. Users generally have a really
high standard concerning video playback. We mentioned in Chapter 2 that some
important factors that influence the video experience are: high visual fidelity,
fast start-up, fast reactivity to user interaction, trick mode support, et cetera.
The relative importance of the factors are user-specific, meaning that some users
have higher expectations than others. When only looking at the user side of
media streaming, adapting to the user’s preference is important. Other factors
like changing network conditions are another reason for adaptive streaming.
Changing network conditions occur on every network connection. Especially
when using mobile Internet, which has become really popular recently, changing
network conditions are a common occurrence. Playing videos under variable
network conditions without adjusting to them, would introduce lots of buffering
and would eliminate the possibility to promptly react to user interactions.

Adaptive streaming is part of the DASH mindset but the functionality has to
be implemented by the DASH client. The DASH standard only provides a stan-
dard way of delivering the information in a way that makes adaptive streaming
possible. DASH provides this information in the MPD. In Section 3.2 we learned
that it is possible to switch between representations within one adaptation set.
These representations can be various media streams encoded at different bitrates
or resolutions. When a media stream has a low bitrate or resolution it will re-
quire less bandwidth to be transferred over the network. The opposite is also
ture, a high quality video will require more bandwidth to be transferred. Another
enabler of adaptive streaming is the HTTP protocol. A DASH client can use
HTTP GET requests to receive segments only when needed. This means that
adapting to changing conditions results in requesting the following segments
from another representation that can handle the new conditions.

We will now give a concrete example of how the adaptive streaming works.
We will study the MPD example in Listing 3.7. We can see that this MPD has
only one adaptation set. In this case we will only talk about the switches inside
an adaptation set, so-called representation switches. We expect basic knowledge
of the MPD syntax; See Section 3.2 for more information, if needed. For this
example, the most important information are the two bandwidth requirements,
segmentAlignment and the segments. We can see that Representation 1 has
a bandwidth requirement of 507218, while Representation 2 has 759159. This
means we will need more available bandwidth to fetch segments from Represen-
tation 2 than from Representation 1. Furthermore, we can see that within the
adaptation set segmentAlignment is set to true. Lastly, we see that each seg-
ment has a duration of 2 seconds (duration/timescale). When a DASH client
wants to use this MPD, he will first decide which representation he will be able
to handle given the currently available bandwidth as well as other factors. For
the purpose of this example, let us assume that the client has enough available bandwidth to handle Representation 2. The DASH client will start fetching the segments one by one whenever he needs more data. If playback and bandwidth conditions would not change while playing the video, the DASH client would be able to stream the entire video at this rate. However, when the available bandwidth would suddenly drop, the client would be obligated to switch to a lower representation to avoid freezing of the video. In particular, the client would switch to Representation 1 by starting to fetch segments using the segmentURL inside Representation 1. For the DASH client not much has changed, segments are downloaded using different URLs and require less available bandwidth but represent a lower quality. When bandwidth conditions change again during playback, the same process can happen again to choose an other representation.

```
<Period duration="PT0H1M23.50S">
  <AdaptationSet segmentAlignment="true" bitstreamSwitching="true" maxWidth="2400" maxHeight="800" maxFrameRate="30" par="2400:800">
    <Representation id="1" mimeType="video/mp4" codecs="avc1.640028" width="2400" height="800" frameRate="30" sar="1:1" startWithSAP="1" bandwidth="507218">
      <SegmentList timescale="15360" duration="30720">
        <Initialization sourceURL="template_init.mp4"/>
        <SegmentURL media="2400x800_500/segment_1.1.m4s"/>
        <SegmentURL media="2400x800_500/segment_1.2.m4s"/>
        <SegmentURL media="2400x800_500/segment_1.3.m4s"/>
        <SegmentURL media="2400x800_500/segment_1.4.m4s"/>
        ...
      </SegmentList>
    </Representation>
    <Representation id="2" mimeType="video/mp4" codecs="avc1.640028" width="2400" height="800" frameRate="30" sar="1:1" startWithSAP="1" bandwidth="759159">
      <SegmentList timescale="15360" duration="30720">
        <Initialization sourceURL="template_init.mp4"/>
        <SegmentURL media="2400x800_750/segment_2.1.m4s"/>
        <SegmentURL media="2400x800_750/segment_2.2.m4s"/>
        <SegmentURL media="2400x800_750/segment_2.3.m4s"/>
        <SegmentURL media="2400x800_750/segment_2.4.m4s"/>
        ...
      </SegmentList>
    </Representation>
  </AdaptationSet>
</Period>
```

Listing 3.7: Part of an MPD showing multiple representations

To be able to adapt to playback conditions, clients have to implement certain algorithms to handle this adaptive streaming. In this thesis we call these algorithms quality adaptation logics. A quality adaptation logic is a collection of rules and logic which determines how and when representation switches should occur.

However, it is not trivial to have the “best” solution for adaptive streaming. There are a lot of factors that influence the way a DASH client will adapt to changes. Various implementations of such quality adaptation logics exist. In this thesis we will only consider simple adaptation logics that change quality depending on only two parameters: available bandwidth and user preference. More advanced adaptation logics will take more parameters into account [52].
but fall outside the scope of this thesis.

3.7 Quality of Experience [12] [46] [4] [1]

Quality of Experience (QoE) is a term used to denote the user satisfaction during a service consumption. QoE in a media streaming context refers to the subjective quality the user experiences when viewing and listening to a media stream. In Section 3.1 we mentioned that users have different background and expectations related to media streaming. This fact correlates directly with QoE because it refers to the quality of a media stream as subjectively perceived by the consumer. According to [61] we can define QoE as “the degree of delight or annoyance of the user of an application or service”. We can see that QoE goes beyond just media stream quality. QoE can be measured in 3 steps:

1. The quality of the media stream at the source,

2. Quality of Service (QoS), which is the quality of delivery of the content over the network,

3. Human experience of the media stream.

Measuring the first two steps is generally fairly easy. The quality of the media stream at the source can be identified for example by the bitrate, the resolution and the framerate. QoS can be measured by monitoring the network connection. Measuring human experience however is not easy. Human perception is usually measured by a Mean Opinion Score (MOS) [46]. The MOS is expressed on a 5 point scale, where 5 = excellent, 4 = good, 3 = fair, 2 = poor, 1 = bad. One way of measuring this is asking the opinion of multiple test subjects. Another way is to do an objective assessment of the human perception using a model that simulates the human senses. This way we will measure the quality in a automatic, quantitative, and repeatable way. Objective assessments have the benefit of not requiring test subjects and thus reducing the cost of the assessments.

We now know what QoE means and how we can measure it, but why do we need to know the Quality of Experience of a certain media stream? QoE represents a standard way of describing the quality of a media streaming session and can be used to compare similar applications. QoE gives a clear indication of possible shortcomings which can then be improved.

DASH was created with high user expectations in mind. It goes without saying that Quality of Experience considerations are included in DASH. Adaptive streaming is one way of meeting the users’ expectations, for example by having fast start-up, low buffer times, et cetera.

The QoE while using DASH is influenced by some factors. The most important once are: buffer overflow and underflow, frequency and amplitude of representation switches and objective quality of the media content. Buffer overflow means having too much data in the buffer, which renders the adaptive streaming less reactive. Buffer underflow means that there is not enough data to keep playing smoothly, eventually causing playback to freeze. The frequency and amplitude of representation switches is an important metric because users generally do not appreciate huge quality switches [21]. Barely noticeable quality
switches are acceptable but users do not want them to be too frequently. The objective quality of the media content can be described with the media stream parameters. Each representation in an adaptation set generally has different objective quality.

DASH has a built-in optional feature to do QoE reporting. The standard defines a set of quality metrics that a client can measure and send to a reporting server. The set contains metrics for:

- TCP connection.
- HTTP request/response transactions.
- Representation switch events.
- Buffer level. This metric is meant to measure the occupancy level of the buffer during the playback.
- Play list. This metric is meant to log a list of playback periods in the measurement period. Each playback period is the time interval between a user action and a stop event. Possible stop events are: the next user action, the end of playback, or a failure that stops the playback.

We first explained what QoE means and how we can measure it. Next we had a brief look at why we would need to measure QoE. Finely we talked about the correlation between QoE and DASH. We will use the information we learned here to do measurements for the use cases in chapters 6, 7, 8 and 9.

3.8 DASH implementations

In the previous sections we emphasized that most adaptive streaming functionality is located in a DASH client. The MPD gives the information for a DASH client to adaptively stream media content. In this section we will take a look at a few client implementations that already exist. Next we will mention how a DASH dataset can be generated.

Firstly, there is a VLC plugin that offers DASH client functionality, developed by the Institute of Information Technology (ITEC). VLC is a free and open source cross-platform multimedia player and framework that plays most multimedia files. Secondly, there is another multimedia player that can act as a DASH client, called OSMO4 which is part of the GPAC project. Thirdly, there are some libraries that offer DASH functionality but are not bound by a multimedia player, namely Libdash, an open-source C++ library, and dash.js, an open-source javascript library. Libdash provides an object-oriented interface to the MPEG-DASH standard and has been developed by bitmovin. Bitmovin also implemented a client application on top of this library called Bitdash. dash.js provides a web-based solution for streaming MPEG-DASH content. The dash.js implementation is an initiative of the Dash-Industry-Forum (DASH-if). It uses the Media Source Extensions (MSE) API as a bridge between DASH content and the HTML5 video element. In Chapter 4 we will take a closer look at the browser technology used for creating a DASH client, including MSE functionality.
3.9 DASH tools

In Section 3.8 we mentioned a few ways of getting a working DASH client, but how can we generate DASH content? There are a few tools available that can give a helping hand. To begin with there is MP4BOX which is also part of the GPAC project [64]. This tool provides the functionality to create a DASH dataset starting from multimedia content. For example, we can specify multiple videos and audio files as input and the tool will create segments and an MPD to describe the multimedia content as a DASH dataset. To use this tool, we have to provide different qualities as input, to make adaptive streaming possible. To create multiple qualities of for example a video we can use ffmpeg, a complete, cross-platform solution to record, convert and stream audio and video [23]. It would be great if there would be a single tool to create a DASH dataset that provides more than one quality for a media stream. ITEC tried to make this tool and called it DASHEncoder. However, DASHEncoder only provides basic functionality to create a dataset starting from a config file. The tool was promising but it is not maintained anymore. Next, there is Nginx-rtmp-module an extension for the Nginx HTTP server [7], which supports generating MPEG-DASH live streams. The dataset it creates can be played with Bitdash and a modified version of dash.js [6].

When creating a DASH dataset we also need to be conform with the DASH standard. ITEC provides a validation tool to check if the MPD is valid with the DASH standard [38]. It will check if the structure is conform and whether the restrictions introduced by the profiles are met.
Chapter 4

Videos in webbrowser enviroments

4.1 HTML5 Video [57]

In Chapter 2 we mentioned that prior to the introduction of HTML5, videos in a web page could only be played because of browser-specific implementations or plugins, for example Adobe Flash. The HTML5 <video> element provides a standard way to embed video into a web page.

HTML5 video is supported in all of the well known browsers. Internet Explorer 9+, Firefox, Opera, Chrome and Safari all support the <video> tag. Nevertheless, the support for some features of the HTML5 video is implementation-specific.

HTML5 video currently supports three video formats: MP4, WebM and Ogg. Table 4.1 overviews browser support for each HTML5 video format. The most important things we need to remember from this table are: MP4 has the best support and Chrome, Firefox and Opera 25+ provide support for all three formats.

<table>
<thead>
<tr>
<th>Browser</th>
<th>MP4</th>
<th>WebM</th>
<th>Ogg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Chrome</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Firefox</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Safari</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Opera</td>
<td>✓ (25+)</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: Browsers and their support for MP4, WebM and Ogg video content on desktop platforms.

In Listing 4.1 we can see how we can include a HTML5 video within a web page. When we define a HTML5 <video> element, we will very likely also define its width and height, which will have direct effect on the size of the corresponding DOM element. When defining a video tag we can specify optional attributes like controls, autoplay. The attribute controls displays a bar under the video with video controls, such as play and pause. Autoplay means
the video will start playing as soon as it has enough data.

Including these attributes while creating a HTML5 video provides an easy way to define the functionality we want for this video element. A video will most likely have a source, this can be specified by the `<source>` tag. When creating a video element we can specify multiple sources, the browser will then choose the first source element it supports. Only when the browser does not support any of the sources, it will display the text specified inside the video tag.

```
<video width="320" height="240" controls autoplay>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.webm" type="video/webm">
  <source src="movie.ogv" type="video/ogg">
  Your browser does not support the video tag.
</video>
```

Listing 4.1: Usage example of HTML5 video within a web page

HTML5 defines DOM methods, properties and events for the HTML5 `<video>` element. Using these methods and properties a client can interact with a HTML5 video, for example to play and pause. The events can be used to catch events related to the HTML5 video, for example when a video can play or has ended. In Table 4.2 we listed some methods, properties and events that will be important for the remainder of this thesis.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load()</td>
<td>Re-loads the audio/video element</td>
</tr>
<tr>
<td>play()</td>
<td>Starts playing the audio/video</td>
</tr>
<tr>
<td>pause()</td>
<td>Pauses the currently playing audio/video</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffered</td>
<td>Readonly. Returns a TimeRanges object representing the buffered parts of the</td>
</tr>
<tr>
<td></td>
<td>audio/video.</td>
</tr>
<tr>
<td>currentTime</td>
<td>Read and write. Sets or returns the current playback position in the audio/</td>
</tr>
<tr>
<td></td>
<td>video (in seconds)</td>
</tr>
<tr>
<td>duration</td>
<td>Readonly. Returns the length of the current audio/video (in seconds)</td>
</tr>
<tr>
<td>ended</td>
<td>Readonly. Returns whether the playback of the audio/video has ended or not</td>
</tr>
<tr>
<td>error</td>
<td>Readonly. Returns a MediaError object representing the error state of the</td>
</tr>
<tr>
<td></td>
<td>audio/video</td>
</tr>
<tr>
<td>paused</td>
<td>Read and write. Sets or returns whether the audio/video is paused or not</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>canplay</td>
<td>Fires when the browser can start playing the audio/video</td>
</tr>
<tr>
<td>ended</td>
<td>Fires when the current playlist ended</td>
</tr>
<tr>
<td>error</td>
<td>Fires when an error occurred during the loading of an audio/video</td>
</tr>
<tr>
<td>playing</td>
<td>Fires when the audio/video is playing after having been paused or stopped</td>
</tr>
<tr>
<td></td>
<td>for buffering</td>
</tr>
<tr>
<td>timeupdate</td>
<td>Fires when the current playback position has changed</td>
</tr>
</tbody>
</table>

Table 4.2: A list of some HTML5 video methods, properties and events and their explanation.
An HTML5 video element can in essence be used as a video decoder. The basic process can be seen as taking a video file, decoding it and visualizing it on the screen. Once a part of the data is decoded and ready to play, a video element can start playing. A video will start decoding a video file even if it is hidden in the DOM. This way we can use the decoded video images and perhaps show them using a different visualizer.

We saw in this section that HTML5 video provides a standard way of embedding video in a web page. The HTML5 video element is already supported by the majority of contemporary desktop browsers, which means we do not require plug-ins to play video in most browser environments anymore. The HTML5 video element has a lot of features and is easy to use as a developer. In the use case chapters 6, 7, 8 and 9 of the thesis we will use the HTML5 video element as a decoder of video data.

### 4.2 HTML5 Canvas

An HTML5 canvas element can be used to draw graphics on the fly. It is displayed as a rectangular element in a webpage. The canvas is in essence a 2D grid, with the top left coordinate being (0,0). We have to use Javascript to draw the graphics on the canvas, and we can draw text, lines, gradients and images. An HTML5 canvas can have multiple graphics on top of each other, for example an image background that is overlaid with a text. An HTML5 canvas provides a flexible way for this drawing functionality.

In Listing 4.2 we see an example of how we can draw an image on a canvas. Here we can see that we first call `getContext("2d")`. This context is a built-in HTML5 object, which has many properties and methods for drawing paths, boxes, circles, text, images, and more. Once we have the context we can draw an image on it by calling the `drawImage` function. Here we can specify what the source is and where we want to draw it in the canvas. In this case we will draw it starting from coordinate (10,10).

```javascript
1. var canvas = document.getElementById("myCanvas");
2. var context = canvas.getContext("2d");
3. var image = document.getElementById("scream");
4. context.drawImage(image, 10, 10);
```

**Listing 4.2:** Drawing an image on a canvas

For the purpose of this thesis we will mostly use HTML5 canvas together with HTML5 video. We will draw images on the canvas for every decoded video frame. When we do this at the same speed as the video frame rate, we can display the video on the canvas. In this case we typically hide the video from the DOM, and only show the canvas. In Listing 4.3 we can see an example of how we can draw a video on a canvas. We use the setInterval function to draw a frame of the video on the canvas at the given rate. To hide the video, we use adjust the css of the video by using jQuery. The HTML5 canvas enables us to manipulate the video data. If we were to use the HTML5 video, this would not be possible. In the use cases in chapters 6, 7, 8 and 9 we use this method to be able to visualize more advanced features compared to flat video footage.

```javascript
1. var video = document.getElementById("myVideo");
2. $(video).css("display", "none");
```
```javascript
3 setInterval(function ()
4 {
5     var canvas = document.getElementById("myCanvas");
6     var context = canvas.getContext("2d");
7     context.drawImage(video, 0, 0, canvas.width, canvas.height);
8 }, 1000/25); // frame rate
```

Listing 4.3: Drawing a hidden video on a canvas

### 4.3 Media Source Extensions [72]

The Media Source Extensions (MSE) API specifies a way to dynamically construct a media stream using Javascript. The HTML5 video tag lacks the ability to dynamically construct a media stream, the entire source has to be known beforehand to be able to decode the video file. The MSE API provides constructs to pass media segments to an HTMLMediaElement, which are the HTML5 video and audio elements. We will only look at the HTML5 video support from MSE. The MSE API has a buffering model that facilitates use cases like adaptive streaming, ad-insertion, time-shifting, and video editing. The MSE API is designed with some goals in mind, for our purpose the most important goals are:

- not require support for any particular media format or codec.
- minimize the need for media parsing in JavaScript.
- allow JavaScript to construct media streams independent of how the media is fetched.
- define a splicing and buffering model that facilitates use cases like adaptive streaming, ad-insertion, time-shifting, and video editing.

A MediaSource object represents a source of media data for an HTML5 video. The media source object will keep track of its readystate (potential values are closed, open, or ended) and a list of all the sourcebuffer objects. Sourcebuffer objects can be used to add media data to the presentation. A MediaSource object can exist out of one or more sourcebuffer objects, for example one for mp4 video and one for webm video. When creating a MediaSource object, the client can attach it to a HTML5 video by generating an URI for the MediaSource and specifying this as the video’s source URL. When adding a new sourcebuffer object to the MediaSource, a client has to specify what media format the sourcebuffer will be handling (e.g., “video/mp4”).

Once a media source is attached to the HTML5 video and the necessary sourcebuffer objects are created, we can start appending data to the sourcebuffers. Once enough data has been added, this data can start playing. When playing a HTML5 video, the client will fetch media data whenever it is needed during playback. When all the data is present in the media source, we can signal end of stream to the media source. This will change the readystate of the media source object to ended and will not allow for any data to be appended from that point forward.
In Listing 4.4 we see an example of how we can create a media source and how we can append data to it. First we create a new media source object. Next we create a URI for the MediaSource object and add it as a source of the video. Then when the MediaSource is opened, we add a sourcebuffer object that expects MP4 data. Once a sourcebuffer object is created, we can start appending data by calling the `appendBuffer()` function. Once all the data is appended to the sourcebuffer we can close the MediaSource by calling the `endOfStream()` function.

```javascript
1 var mediaSource = new MediaSource();
2 var video = document.getElementById('myvideo');
3 video.src = window.URL.createObjectURL(mediaSource);
4 mediaSource.addEventListener('sourceopen', function(e) {
7 ...  
8 var sourceBuffer = mediaSource.addSourceBuffer('video/mp4; codecs="avc1.640020"');
9 sourceBuffer.appendBuffer(segmentOneOfVideoData);
10 ...  
11 if(!sourceBuffer.updating)
12 sourceBuffer.appendBuffer(segmentTwoOfVideoData);
13 ...  
14 if(!sourceBuffer.updating)
15 this.endOfStream();
16 ...  
17 }, false);
```

Listing 4.4: Example use of MSE

We mentioned earlier that MSE allowed for adaptive streaming use cases because it allows us to append data on a per-segment basis. This means that when we want to create a MPEG-DASH client that has to dynamically download individual segments, we can use MSE to provide these segments to the HTML5 video. MSE together with HTML5 video provide an easy way of displaying segment data received by a DASH client. We will therefore use this method in the use cases in chapters 6, 7, 8 and 9.

4.4 WebGL [62] [47]

WebGL is an API that can be used for producing 3D graphics on the web. WebGL grants the possibility to provide hardware accelerated 3D graphics to a web page. WebGL is based on OpenGL ES 2.0 a subset of the OpenGL API, designed for embedded systems [43].

Before WebGL, it was only possible to provide a true 3D web-based experience via plug-ins like Adobe Flash. WebGL is part of HTML5 but is not an official specification. It is available in most browsers that support HTML5 elements [62]. It is the browser’s decision whether they support WebGL or not. Nevertheless, WebGL is a cross-platform solution, with the ability to run on devices ranging from smartphones to desktops.

3D rendering with WebGL uses the HTML5 canvas element to visualize the rendered result. Just like drawing on a 2D canvas, rendering in 3D is done via Javascript functions. In Section 4.2 we learned that when drawing on a canvas, we use a context for the 2D drawing. This context provides properties and
methods to draw on this 2D canvas. For 3D rendering we will use a drawing context specific to WebGL. This context defines functionality specific for WebGL in contrast to the 2D context described in Section 4.2. Because it uses the HTML5 canvas, WebGL can directly be displayed inside a web page.

The WebGL API provides some low-level methods to manipulate 3D graphics, similar to OpenGL. Just like OpenGL, WebGL has a rendering pipeline. We can program certain sections of the pipeline by writing shader programs [14]. Each shader will have a set of inputs and outputs that are related to the involved pipeline section. A simple example of a shader would be a fragment shader that changes the color of a picture.

WebGL enables a variety of applications. In our context, we will use it to manipulate video data. At first it may not be clear how WebGL can manipulate video data, but in essence video is just a set of images shown at a fast rate. We can manipulate the images separately, to have the same result as manipulating a video. One application for this is the ODV Viewer described in Section 4.5. Another application can be found in the use case in Chapter 8.

### 4.5 ODV Viewer

The use cases in chapters 6, 7, 8 and 9 are created with special focus on Omni-Directional Video (ODV). ODV is video with a 360 degree field of view. Such a video will give a different experience compared to a regular video. When viewing an ODV we will get a sense of the captured surroundings. This will increase the feeling of presence in a remote environment [27]. In an ODV we generally see more information because we cover a more broad field of view. We will be able to see dynamic events that would otherwise fall outside of the camera view. Figure 4.1 shows a video frame of an ODV. We can clearly see that this video has a larger field of view compared to a classic video-clip.

![Figure 4.1: A video frame showing the 360 degree field of view of an ODV.](image)

Because we are familiar with viewing a standard video, we generally prefer watching some part of the ODV, and pan (rotate) to see other parts of the ODV. In other words, always provide only a limited view window into the ODV content. This kind of video viewing is barely supported by standard video players, so for the purpose of the use cases in chapters 6, 7, 8 and 9 we will use an ODV Viewer implementation made at EDM.

The ODV Viewer is a web-based implementation that supports displaying and interacting with an ODV. The viewer uses a HTML5 canvas to visualize the video. The ODV viewer provides a uniform way of interacting with a canvas using a spherical (3D) and a planar (2D) representation. The spherical renderer
is created in such a way that the ODV will be mapped on a sphere using WebGL. The ODV viewer allows for interaction with the canvas to look around and zoom. When viewing the video with the ODV viewer, we will only see a part of the video, called the viewport. When looking around or zooming, this viewport will change. Figure 4.2 shows the video footage inside multiple viewports. By panning or zooming the footage within the original viewport changes. The ODV Viewer has a planar renderer, to display 2D videos but still have the interaction in the same way as the 3D case. When using a planar renderer, we also see a viewport on the screen. This viewport can display the entire 2D video or just a part of it. If only a part is visible, the panning interaction can be used to view other areas of the video. In Figure 4.3 we see an example of how a 2D video can be displayed using the ODV Viewer. We can also see what result the zooming interaction has on the ODV Viewer.

By using the ODV Viewer we have a web-based solution that can display standard 2D video and ODV and have a general way of interacting with the videos. Therefore, we will use the ODV Viewer for the visualization of the use cases in chapters 6, 7, 8 and 9.

Figure 4.2: Screenshots of the ODV Viewer while using a 3D spherical renderer.
(a) An example of how a 2d video can be displayed within the ODV Viewer.

(b) The ODV Viewer enables us to interact with the video. In this case we zoomed in on a part of the video footage.

**Figure 4.3:** Screenshots of the ODV Viewer while using a 2D planar renderer.
Part II

Use cases
Chapter 5

MPEG-DASH Implementation Framework

5.1 Introduction

In this thesis we aim to create a number of DASH-related use cases. To realize the use cases described in chapters 6, 7, 8, and 9 we made use of an implementation framework provided by EDM. The implementation framework is designed with focus on reusability. It exists out of two core components, a backend and a frontend.

The **backend** component is a cross-platform component that provides common DASH client functionality which is likely to be required in a general DASH client or in typical DASH-related use cases. Because the backend component has all this functionality it is beneficial to create it as a cross-platform component. To achieve this, the backend is written in Node.js. Node.js is a platform built on Chrome’s Javascript runtime [29]. Node.js applications are written in Javascript and generally focus on server-side processing and networking. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices [58]. In Section 5.1 we learned that a DASH client is responsible for fetching segments and handling the adaptive streaming logic related to this. In this implementation framework, the backend will contain this functionality. In Section 5.2 we will take a more detailed look on the structure of the backend and the functionality it provides.

The **frontend** component is a platform specific component that provides a graphical user interface for the DASH client. We want to be able to create customized frontends that are based on the same general DASH functionality. To achieve this, we will connect every frontend with a backend, so that every frontend can use the generic backend functionality. This way we are able to make frontend components that have unique functionality in addition to this general DASH functionality. This relates exceptionally well to creating multiple DASH-related use cases because we can just create a different frontend for each use case. Every frontend will communicate with the backend to eventually receive DASH segment data. Some basic backend/frontend communication is necessary to implement a minimal frontend. The rest of the communication can
be used to create custom functionality in every frontend.

In Figure 5.1 we see a visualization of how the framework can be used. Multiple platform-specific frontends can connect to the backend. As always, the DASH data is hosted on a HTTP server. The backend will use HTTP requests to receive data from this external server. After some basic backend.frontend communication, the backend will be able to relay this media data to the specific frontend. In a normal scenario, the frontend, the backend, and the content server will be hosted on different machines. However, it is possible that they are all hosted on the same machine.

![Figure 5.1: The structure of the DASH implementation framework.](image)

For the use cases in chapters 6, 7, 8 and 9 we will create separate frontend components that are all grounded on the functionality provided by the backend component. Some client-side functionality can be generalized for multiple frontends, which encourages the use of reusable libraries for such functionality. In Section 5.3 we will talk about libraries that provide this general frontend-side functionality. However, dedicated frontend-specific functionality will be implemented for each use case to account for the wanted result.

In the use cases in chapters 6, 7, 8 and 9 each frontend has to connect to the backend. After connecting, some configuration options have to be selected. In Section 5.4 we will explain what configuration options are currently available in the backend.

In addition to creating some general client-side libraries, some functionality will be used from external libraries. In Section 5.5 we look at what external libraries are used and what functionality they offer.

### 5.2 Backend software architecture

The backend component is created as a streaming server of DASH content to a number of frontends. The backend component itself is built out of several modules that take care of specific tasks. In Figure 5.2 we can see a representation of the software architecture of the backend. In this section we will provide an explanation for each of these modules and describe the relation between multiple modules.

The backend will have to communicate with multiple frontends. To do this, it requires a module to handle incoming and outgoing communication related data. The **CommunicationManager** will communicate with a frontend component
The modular software architecture of in the backend.

Through messages. These messages are formatted in the Javascript Object Notation (JSON) format [41], which is a standardized format for representing data and an alternative to XML. JSON is often used in the case of exchanging data between servers and web applications. JSON is less verbose compared to XML while still being readable for humans. The backend also provides the ability to use a different format for the messaging, if the JSON format would be insufficient.

The CommunicationManager module deals with two categories of messages: control messages and data messages. Each message category has its own channel of communication. For the control messages, a HTTP server is used to provide a HTTP request and HTTP response communication channel. For data messages, there are currently two alternatives, a TCP connection and a websocket connection. Both are used as a one way channel for the backend to send data messages to the frontend. Control messages are sent from frontend to backend and provide “control” information. There are multiple types of control messages that trigger specific functionality, for example registering a frontend or starting the playback.

Data messages are sent from backend to frontend and contain media data. There are also multiple types of data messages: a media data message that contains media data about the currently active stream, a meta data message that communicates meta data about the active stream and an end of stream message to inform the frontend that no more data will be arriving for the active stream.

Another important backend module is the MediaSceneGraph, which creates a graph representation from a media experience. A media experience is a set of one or multiple MPDs where MPDs can be related to each other. These dependencies can vary from streams that should be played after each other, to streams that must be played in parallel. When the MPD has no dependencies, the graph will be created with just one node. A frontend will have the ability
to choose what media experience it wants to play. Once the choice is made, the MediaSceneGraph module will construct a graph for the chosen media experience. The MediaSceneGraph module will collaborate with another module, the MediaStreamRepo. This module will hold data for all the media experiences that are offered by the backend and parse the MPDs that belong to these media experiences. The MediaSceneGraph will get media experience data directly from the MediaStreamRepo, which holds all the necessary data to construct the media scene graph.

The backend will stream media data to the frontend through data messages. Because the backend provides the ability to play an entire media experience, it is possible to have dependencies between MPDs. When “playing” a media experience, there is always at least one active stream. A stream is active when it should be played and visualized by the frontend. The backend keeps track of the active stream(s) in the PlaybackLogic module. The PlaybackLogic module makes sure segments are fetched for these streams. The PlaybackLogic receives incoming messages from the CommunicationManager. According to the received message, functionality from other modules will be used. For instance, when a media experience message is received, the correct functionality from the MediaSceneGraph is used.

Adaptive streaming is part of the backend but is split into multiple modules. First there is the DownloadManager, which handles the downloading of segments per media data stream. The DownloadManager module stores information about what segment indices have already been downloaded and which segment index is the next in line. In Section 3.6 we explained that there are various algorithms on how to determine from which representation the next segment should be chosen. This functionality can be implemented through the QAdaptationLogic module. The QAdapationLogic is defined in an extensible way to make the integration of various quality adaptation algorithms possible. Segments are downloaded on-the-fly in a DASH client, to account for this functionality the module SegmentDownloadRateEnforcer is created. The SegmentDownloadRateEnforcer determines when segments will be downloaded according to buffer levels and other factors.

We explained earlier that a media experience is a set of one or multiple MPDs with possible dependencies between these MPDs. With this functionality in mind, a module called PrecacheLogic was created. When we have a dependency between two MPDs where for instance one has to play after the other, it can be helpful to already download a part of the second media data stream while the first one is still active to reduce start-up time for the second stream drastically. In this case it would be possible to start playing the second stream immediately after the first stream ends. We call this functionality precaching and it is located in the PrecacheLogic module. The backend is flexible enough to support various precaching algorithms that determine how much and when the data should be precached. When exploited intelligently, this functionality can lead to playing multiple streams after each other with no buffering occurring in between. In essence, we can create a seamless “path” between multiple nodes. In the use case in Chapter 6 we will give extra attention to this concept.

Each frontend has to choose precaching and quality adaptation logic it wishes to use. In Section 5.4 we will look at what logics are currently available and how a frontend can configure the backend to use a specific configuration.
5.3 Frontend libraries

In this thesis we will create multiple web-based DASH use cases that each use the backend described in Section 5.2. We will encapsulate each use case into a different frontend component. Each frontend component will have some unique functionality to account for the result we want to accomplish. However, some parts of the client implementation will be the same for multiple frontends. To make this functionality reusable, we created libraries that abstract these capabilities. Because we are working on a web-based platform, all libraries are written in Javascript. The following libraries have been developed: a communication library and a visualization library.

Figure 5.3: The communication library abstracts the communication between a web-based frontend on the one hand and a backend and gesture-tracking server on the other hand.

The communication library implements the backend-frontend protocol at client side, and hence allows a web-based frontend to communicate with the backend. It is split into two modules: BFCommunication and GTServerCommunication. In Figure 5.3 we can see that the BFCommunication module handles communication between a HTML5 frontend and backend. Both the control messages and data messages will be handled by this module. It provides the ability to receive both categories of messages and only send control messages. We will not be able to send data messages because that channel is meant for the backend to push data to the frontend. When the BFCommunication module receives incoming message data, it will expose it to the frontend through events. A frontend will always have to listen to the type of events it is interested in. This way we can choose how we want to handle different types of messages for each frontend but still provide the ability to receive all messages in every frontend. This module also enables a frontend to send every type of control message that is currently supported by the backend. The control channel will use HTTP requests and responses, because of this it will always be the frontend that sends a control message and then afterwards receives a response from the server containing a control message. As a data channel we will use a websocket connection. The websocket specification defines a full-duplex single socket connection over which messages can be sent between client and server. The websocket standard simplifies much of the complexity around bi-directional web communication and connection management [74]. Even though websockets offer a full-duplex chan-
nel, we will only use it as a channel for the backend to send data messages to the frontend. In Figure 5.3 we can also see that the GTServerCommunication module will handle communication between a frontend and a gesture-tracking server. The gesture-tracking server is used to recognize gestures and allow users to interact with frontends using gestures. The basic gesture tracking communication functionality as well as the gesture-tracking server were provided by EDM. We added extra functionality that will be used in the use case in Chapter 9 and remodeled the structure of the module so that it works in the same way as the BFCommunication module. Just as the BFCommunication module it will use events specific to each message to signal incoming data to frontends. Communication between gesture-tracking server and frontend will also be split into two channel, a control channel and a data channel. For both channels the gesture-tracking server provides websocket connections. Generally, the gesture-tracking server will only receive and send control messages. Only in use cases comparable to the one in Chapter 9 data messages will be send over the second channel.

![Diagram](image)

**Figure 5.4:** The visualization library structure

The visualization library can be used to visualize incoming DASH media data on a webpage. This library exists out of a Viewer module and a ViewerToOdv module. In Figure 5.4 we can see the structure of this visualization library. The Viewer module will create a cohesion between a MediaSource object and a HTML5 video, while the ViewerToOdv module can be used to display the content of that viewer on an ODV Viewer. The Viewer module contains basic functionality to append DASH media data to a sourcebuffer of a MediaSource object. The Viewer module will use a HTML5 video to visualize this data. Because we are handling videos, we should be notified when a video has buffered enough data to be able to play. The Viewer module will emit a canplay event when this is the case. The Viewer is meant to play DASH media data. In this case being able to play a DASH stream means that the video can play and has reached a buffer level higher than the minimal buffer time specified in the MPD. When the end of a video is reached, the viewer can signal end of stream to its MediaSource object. From that point forward we will not be able to append more data to the MediaSource. The ViewerToOdv module will always be used in collaboration with a viewer. It will use the content in a viewer and visualize it in an ODV Viewer. In the use cases in chapters 6, 7, 8 and 9 we will mostly work with ODV content, so it is beneficial to provide this functionality as a general module.

5.4 Configuration options

After a frontend connects to the backend, some configuration options have to be selected. These configuration options are sent from the frontend to the
backend using control messages. For the configuration of the media experience, the precaching logic and the quality adaptation logic multiple possibilities are available.

The backend has some media experiences available. These media experiences describe the mutual relation between MPDs. In the implementation framework, we use these media experiences to play video streams. When a media experience is chosen in the frontend, data will be received for this experience. For this thesis, we created various media experiences, all describing some unique feature. Most of these media experiences are use case specific.

A precaching logic determines when the future stream will be precached. The implementation framework currently contains two precaching logics, both are developed by EDM. We precache data for future streams while the active stream is playing to be able to seamlessly switch to one of the future streams. The first precaching logic, precaches the future streams at the beginning of the active stream. The second precaching logic precaches during the playback of the active stream. This precaching logic uses a parametric approach. In the current implementation, when the playback of the active stream reaches 75% of its duration, the precaching of future streams will start. The current implemented precaching logics are fairly simple but provide sufficient functionality to be able to seamlessly switch to a future stream.

A quality adaptation logic determines from what representation segments will be fetched. Currently, the implementation framework provides three quality adaptation logics:

- Always Lowest
- Always Highest
- Best Fit

The always lowest and always highest respectively fetch the lowest and highest quality for each segment. The best fit quality adaptation chooses a representation according to the available bandwidth. This means that a quality will be chosen that can be played with the current bandwidth condition. All these quality adaptation logics are provided by EDM. In the use case in Chapter 7 we will create an other quality adaptation logic to provide functionality for this use case.

Different configuration options result in different outcomes in the frontend. A simple example of this is the difference between the always lowest and always highest quality adaptation. A user will respectively see a low or high quality video stream. The configuration options have to be carefully chosen when specific results want to be achieved.

5.5 External libraries

For some client-side specific functionality, external libraries will be used. The existing libraries offer the required functionality for the use cases in this thesis.

In the use case in Chapter 6 we will draw a graph on the webpage. To draw these graph we will use the Raphaël Javascript library. Raphaël provides functionality to simplify drawing vector graphics on the web.
When the frontend receives data messages from the backend, the actual media data is encoded in a base 64 string. To decode this base 64 string, we use base 64 decoding functionality provided on the Mozilla Developer Network [55]. We will use the functionality to convert a base 64 string into a decoded uint8 byte array. We also considered using the built in “btoa” function to do this, but this solution was slower compared to the other solution.

To be able to write a shader program for WebGL, we use a utility library that simplifies some WebGL functionality, called webgl-utils. WebGL was not the focus of this thesis, we therefore followed a tutorial to be able to account for simple WebGL functionality [29]. In these tutorials the webgl-utils was used. This library reduces the amount of work to be done, to be able to write shader programs. Therefore it seems like an obvious choice to reuse this library to write the shaders used in the use case in Chapter 8.
Chapter 6

Path-Based Non-linear Video

6.1 Introduction

In this use case, we provide a different video experience compared to regular video streaming. In regular video streaming, also called linear video streaming, we play the same sequence of the video frames in each streaming session. This means that when we watch a linear video multiple times, the video data will not change. Users are familiar with this kind of video streaming because it is widely used. When we watch a movie, a single sequence of video frames will be played starting from the first frame of the movie. Users have almost no interactive possibilities with this video, rather than play, pause, fast-forward, seeking and et cetera.

In contrast to linear video, non-linear video offers users the option to influence the sequence of the video playback [15]. Non-linear video enables a user to interact with a video and personalize its content to his liking. A simple example is the playback of a movie on DVD that lets the user choose the ending of the story [15]. In this example the user creates his own experience for the movie by playing it in that particular sequence. There are a wide range of applications for non-linear video; some more conventional ones are news-on-demand and virtual tours. In Figure 6.1 we see an image of a virtual tour of the Smithsonian national museum of natural history. In this figure we can see that the setting in this virtual tour is the museum. Buttons are located on top of the setting and on the bottom to navigate in this scene. We can use this navigation to view specific parts of the non-linear video. Typically, in a non-linear video only a part of the video experience will be personalized, the rest of the content sequences will be the same for all users. In the movie on DVD example, the first part of the movie will be the same for every user that watches the movie, the personalization only occurs when he has to choose the ending of the story. In applications as news-on-demand, the user is provided more information about subjects that are of interest to him. This information can be provided in for example another content sequence or in textual form. Non-linear video offers media developers the ability to create experiences that would otherwise be impossible to achieve in a linear video.
Figure 6.1: An image of a non-linear video for the Smithsonian national museum of natural history [13].

Linear video has been adopted by broadcast systems [13]. It can be provided as live, on-demand or near on-demand content. Providing the same functionality for non-linear video is more challenging. Because content can vary in non-linear video, it is not as simple as providing one video file, which is the case with linear video.

In this use case we will work with content that embodies a virtual tour. In this tour we will let users choose their own path within the provided video scene. The video content can be regarded as a virtual space, and the path as the navigation through this space. A virtual tour will be a concatenation of distinct parts of the video footage. We can for example let users “walk” through the scene and choose which direction they want to take at an intersection in the scene. There will be a predefined number of directions the user can choose from at each intersection.

In Section 3.1 we mentioned that user expectations concerning video experience are high. These expectations relate mostly to linear video, because this is currently the general approach. When viewing linear video, stalls that make the video freeze are not appreciated. In linear video, we can solve this issue by maintaining a sufficient buffer level at all times. However, in non-linear video this issue is more challenging. In non-linear video, the video player does not know which “path” the user will take. The video player can take an educated guess based on decisions made at previous intersections, but will never know exactly what sequences the remainder of the video will contain. To avoid stalls in a non-linear video, we will have to buffer the beginning of all the sequences a user can choose from at the next intersection. We will call this buffering precaching. Once the choice is made, we should continue buffering only the sequence that was chosen.

In this use case we will provide a non-linear video experience, that puts the focus on choosing the next path at predefined points in the video. These
selection points mostly coincide with physical intersections that appear in the video footage. In Section 6.2 we will take a look at a possible solution that provides this kind of video experience. Afterwards we will propose our solution in Section 6.3 and its implementation in Section 6.4.

### 6.2 Related Work

A quality-adaptive prefetching solution for interactive branched video using HTTP-based adaptive streaming is proposed by [45]. They use quality-adaptive prefetching to precache certain sequences at a quality that can be handled by the current network conditions and other factors. The term prefetching is used to indicate the same functionality as the term precaching, which is defined in Section 6.1. They use the term interactive branched video, which means a non-linear video that has certain points where a user can choose his path. They describe a model and implementation of a solution. The model will determine what sequences have to be precached and when this precaching has to occur. They provide multiple prefetching policies to address this.

Their solution is based on HTTP-based Adaptive Streaming (HAS), which uses a similar structure as MPEG-DASH as explained in Section 3.1. The solution uses a manifest file with information about multiple segments or chunks. To provide the ability to define non-linear video, they provide an additional file containing information about possible branch points. Branch points are intersections within the non-linear video, where a user can choose his path. When defining a branch point, one has to specify four rules: the path the client has to take before coming to this branch point, the navigational options a user has at the branch point, the weight of each path and the message that can be displayed when choosing a path. When playing a video, their player keeps track of what the upcoming branch points are by checking the file.

Their prefetching policies focus on maximizing video quality while still providing uninterrupted playback when switching between sequences. To tackle the prefetching problem they consider it as an optimization problem, which means finding the best solution out of all possible solutions. They want to maximize the quality of the sequence that is currently playing while still prefetching the highest possible quality before the current sequence has finished playing. To prefetch data for future sequences, multiple TCP connection can be opened to stream the data in parallel. The prefetching policy decides whether or not multiple connections are going to be used. The choice is made considering a trade off between overall quality of the video experience and downloading future sequences in time. When downloading future sequences, they create a download order, depending on the likeliness of selecting that path and when the data should be available. The prefetching policies try to cap the amount of data that will be prefetched to avoid downloading too much unnecessary data and thus wasting available bandwidth. Different prefetching policies will have different end results. Some prefetching policies may introduce stalls in the video, if for example the prefetching of segments happens after a branch point.

Depending on the prefetching policy, multiple TCP connections might be opened simultaneously. New TCP connections will only be opened to be able to download enough data in time for each path in parallel. If a connection is idle, the connection can be reused for fetching data from different streams.
Because data is represented as segments, the client has to fetch the data on a per-segment basis. When multiple paths are involved, a round-robin ordering will be used to download segments for each path in an intertwined fashion. This means that the client will request the first segment of each path first. In this iteration the path with the highest weight will be selected to be downloaded first. The prefetching policies will determine possible candidates for schedules. The schedule will tell what segments should be downloaded when, using a certain connection. Out of these candidates, the “best” option should be chosen. The first prefetching policy is “Optimized non-increasing quality”. It will put a limit on the maximum number of connections and will not open new connections in future steps. The quality of consecutive segments will not increase. The non-increasing constrain will still result in a lot of candidates. Consecutive chunk can have the same or a lower quality. Because of this they created a second prefetching policy “Optimized maintainable quality”, which will maintain the quality chosen for each path for consecutive chunks.

They include two other prefetching policies “Single connection” and “Greedy bandwidth”. The former opens only one TCP connection; the latter behaves more bandwidth aggressive by choosing qualities and number of parallel downloads to maximize the number of requested bytes.

In Figure 6.2 an example of the round robin ordering combined with multiple parallel connections is shown. The terms used in this example are different from the terms used in this thesis: a chunk is a synonym for segment, segments are sequences and branch points are intersections in the non-linear video.

![Figure 6.2: Example use of the round robin solution by [45]. The terms used in this example are different from the terms used in this thesis: chunks are segments, segments are sequences and branch points are intersections in the non-linear video.](image-url)
respond with sequences in our terminology, and branch points equal intersections in the non-linear video. At the bottom we see the available sequences, with each sequences having a specific color. In the graph we can see that the first sequence of the non-linear video is sequence e1. When the playback arrives at the first intersection there are three choices: e2, e3 and e4. This means that before the playback arrives at this intersection the client should have precached data for each of these sequences. In the connection part of the figure we can see how the downloading of segments occurs. First the download for segments of sequence e1 started. Once e1 was being downloaded, the chosen prefetching policy decided when the precaching starts to have enough data before the end of e1. To meet this deadline, the prefetching policy made the decision to open new connections. Because a round robin ordering is used, the first segment for all future streams were downloaded. The third connection was opened to meet the download deadline. When a connection is idle at any point during the playback, it is used to download other segments. After all the first segments are downloaded, the client started downloading the all the second segments. Because connection three was idle the last segment of e2 was downloaded. When the user made the choice to go to the path of sequence e3, the client had two segments already available for this sequence. At this point only the last segment had to be downloaded for this sequence, namely segment 9. When segment 9 was being downloaded, the same procedure happened again. Segments that should be played, were added to the playback buffer. At the end of the entire process, the playback buffer contained the user his personalized non-linear video.

As an implementation, they have created a player that can open multiple TCP connections. To play a video, they use a playback buffer which they fill with segments of data. When prefetching chunks, they download them into browser cache to reduce downloading times when they actually require the specific data to be added to the playback buffer. Users are able to modify their path when they reach a branch point. Data relating to the path will then be transferred from the cache into the playback buffer. The installed prefetching policy determines the number of TCP connections that will be opened simultaneously to do the prefetching. Based on the installed policy, the player will actually open the determined amount of TCP connections.

### 6.3 Approach

In our approach we use the implementation framework described in Chapter 5. Because we have a separation of frontend and backend, functionality for playing non-linear video is fragmented over the frontend and backend.

To enable the streaming of non-linear video, we created a media experience file. This media experience file contains a list of MPDs and the relation between them. There is a relation between MPD A and MPD B when we should be able to stream MPD B after playing MPD A. In this approach we can describe each sequence of the non-linear video in a separate MPD file. This means that in a lengthy non-linear video, a clean separation between sequences is still possible because each sequence has its own file. When using only one manifest file like the solution provided by [15], the MPD may get large and contain multiple sequences, which wastes bandwidth and makes the file less surveyable.

From a media experience file the backend MediaSceneGraph module will
create a graph. This graph will have sequences as nodes and relations between sequences as edges. In our approach a sequence will be called a stream. We will have one active stream. The sequences that have to be precached to avoid stalls will be called future streams. This means that when a node in the graph has children, the active stream related to this node has one or more future streams.

In Listing 6.1 we give an example of the structure of a media experience file. We can clearly see that the file is divided into a description of nodes and edges. We see that every node has an mpdURL and an id attribute. These two parameters are the most important because they uniquely describe a node. From this point forward, a node will be identified by its id and not its MPD. Because ids are used to identify a node, they should be unique. In the example we have three nodes with ids ind01, ind02 and ind03. Edges are described by their source and target. To identify a node, we use the id defined in the node section. The listing contains two edges: the first edge connects ind01 to ind02, while the second edge connects ind01 to ind03. In Figure 6.3 we can see how this file will be translated into a graph. We will have a graph with node ind01 as root node. It will have two edges, one to ind02 and the other to ind03.

```json
1 {  
 2     "nodes": [{  
 3         "id": "ind01",  
 4         
 5         "caption": "RootNode",  
 6         "mpdURL": "http://.../template.mpd",  
 7         "duration":24.79,  
 8         
 9       },{  
10         "id": "ind02",  
11         
12         "caption": "Ind Stream 2",  
13         "mpdURL": "http://.../template.mpd",  
14         "duration":24.79,  
15         
16       },{  
17         "id": "ind03",  
18         
19         "caption": "Ind Stream 3",  
20         "mpdURL": "http://.../template.mpd",  
21         "duration":24.79,  
22         
23       }],  
24     
25     "edges": [{  
26         "source": "ind01",  
27         "target": "ind02"  
28       },{  
29         "source": "ind01",  
30         "target": "ind03"  
31       }]  
32 }  
```

Listing 6.1: Example of a media experience file

The frontend provides a GUI for the user to interact with. It will not only include the visualisation of the non-linear video, but also connection and configuration options for the backend. In the frontend we will choose what media experience we would like to play. After choosing a media experience, the back-
end will send the meta data about this media experience. Maybe the most important information in the meta data is information about the currently active stream and its future streams. From this information we know what stream we should display on the screen and what streams are possible paths in the future. A user will be able to interact with the non-linear video in the frontend when he reaches an intersection. The user will have to make a choice which path he would like to follow after this intersection. The frontend will send the chosen path to the backend, identified by its id. This will be sent as a message with type \texttt{AIVIE,BFPROT,SET,FUTURENODE}. From this point forward the chosen stream will be marked as active stream. Once the previous active stream has ended, the new active stream should be played in the frontend. After initiating playback for the new active stream, we first playback the data that has been precached for it. While the playing this data, we start precaching its children and download the rest of the segments of this stream. If the choice is made prior to the end of the active stream, the transition between streams should be seamless. When the user makes a choice after the active stream has ended, the new active stream should start immediately. In this case a stall will happen at the end of the active stream. This stall is completely caused by the user because he did not make a choice before the end of the stream. In this case we will thus define a transition seamless when playback starts immediately after making a choice.

The backend will fetch segments for currently active and future streams and send them to the frontend. When sending data from the backend to the frontend, we include a field identifying to which stream it relates. This means we can filter the data relating to each stream. We will append the data to a viewer that is associated with the involved stream. Each viewer will hence contain only data that pertains to a single stream. The viewer that represents the currently active stream should be played and displayed on the screen. When playing, we will receive data for the active stream and its future streams. When we have received all the media data for the active stream, we will receive an end of stream message. This end of stream message will signal the frontend that it should close the stream because no more media data is coming.

Because we will have a variable number of viewers, we created a class \texttt{ViewerManager} to manage them. This class contains the logic needed to play the currently active stream. It encompasses a data structure that contains each viewer. This data structure provides us easy solution to append data to a specific viewer. The ViewerManager will hold meta data about the media experience, which means the manager knows which stream should be the active stream and which streams are the future streams. The ViewerManager will always play the active stream until the end. If a choice is made by the user, the ViewerManager will replace the active stream with the newly chosen stream,
and start playing it. By doing so, we try to provide a seamless switching between streams. The ViewerManager will start the new stream after the choice is made. If the choice is made after the active stream has ended, the stall is caused by the user. The ViewerManager makes sure that once the user makes a choice, the stream starts as soon as possible.

Precaching logic is located in the backend PrecacheLogic module. This module will take care of what data should be precached and at what time. Currently we have two strategies. The first strategy will download the first few segments of the future streams when the downloading of the active stream starts. This means that we will always be able to start playing the future stream after the active stream ends. The downside is that we limit the initial available bandwidth for the active stream. The other strategy uses a parametric approach. It will decide when to download future streams, taking into account the current playback time of the active stream. Currently the available precaching strategies are limited and could be improved. For purpose of a use case proof-of-concept they are nonetheless sufficient.

In contrast to [45], the quality of the future streams is decided by the QAdaptationLogic that also decided the quality of the active stream. In the current implementation active and future streams are handled the same way. In future implementations the choice can be made to handle them separately. In the current implementation the only separation between active and future streams is the amount bandwidth each stream category is assigned. Currently the backend uses a division of 60% of the available bandwidth for the active stream and 40% for the future streams. In the future this division should be configurable depending on user preferences. Because the implementation framework is still under development, this feature is not yet included. For our purposes the static division will suffice. There are two main disadvantages to this. First, the active stream will only choose a representation that is playable with 60% of the available bandwidth, even when the chosen representation for future streams do not need 40% available bandwidth. Second, when we have a lot of future streams, they will have to divide a limited portion (i.e., 40%) between each other to choose a representation, resulting in a low quality representation per stream.

Every viewer will contain media data for its stream. Because we are handling multiple viewers, we store a lot of data in the browser’s memory. To prevent browser memory from becoming an issue, we could implement a module that determines whether it would be helpful to remove the data of a certain viewer to decrease the memory. For example, a simple logic is to delete every viewer that is already played. We could also implement a more advanced logic that calculates the chance of a specific viewer being played again in the future. If the chance is below a certain threshold, it could delete this viewer. To avoid problems in the backend, only viewers containing a full stream of data can be deleted. For streams that have only precached certain segments, the backend will only send the remaining segments. This means that when we delete these precached segments, we will not receive them again when the backend starts downloading this stream.
6.4 Application

In this section we will describe the application we built for this use case. In Figure 6.4 we see the layout of the application. We chose this layout to have some similarities with the Qt frontend provided by EDM (see Figure 6.5). By adopting a similar layout, users only have to learn the functionality of one frontend and should be able to use the same functionality in the other frontend. We will now look at each object in this layout in detail.

![Figure 6.4: A screenshot of the HTML5 frontend for the choose your own path use case](image)

In the top left corner, we see an ODV Viewer, which displays the active stream. We are able to spatially interact with the ODV Viewer as described in Section 4.5. In addition to this spatial interaction, we provide some basic video controls: play, pause, fast-forward, normal speed. We will use the bar under the ODV viewer to let users know how much data is buffered and what the current playback time of the video is.

Future streams should normally not be displayed but as a proof-of-concept we display them in the bottom left corner. In this case we only provide the ability to have two future streams and only display the first frame of each future stream. Below the visual representation of the future stream we see two buttons for performing path selection at the next decision point.

In the top right corner we have registration and configuration options. There are two checkboxes, a connect and a provide remote dataset checkbox. By clicking connect we will make a connection with the backend. The remote dataset tells the backend we want to use the dataset stored with the backend. The remote dataset will include multiple media experiences, quality adaptation
Figure 6.5: A screenshot of the QT frontend for the choose your own path use case and precaching logics. We present all the dataset options in the corresponding dropdown boxes. In the future, users will be able to provide their own media experiences, but for the time being we will always use the dataset provided by the backend. When a user makes a selection with the dropdown boxes, the choice will be sent to the backend. Because of this, the frontend and backend should always have the same information concerning the chosen configuration options. After the user chooses a media experience as well as a quality adaptation and a precaching logic, the frontend will be ready to start playing.

When a media experience is chosen, the graph on the right will be created. As long as no media experience is selected, the box will be empty. The graph contains a node for each stream and labels each node with its id. This gives the user a notion of the kind of path to expect when playing this media experience. The graph is not only meant as visual support, but is also used as a navigation tool. Each node of the graph is clickable and can be used to make a choice for the path. The user is able to select the node corresponding with the path he wishes to take. The user can only choose nodes that are children of the currently active stream. We use colors to display the status of a node in this graph. Red means the node’s stream has been played and ended, green means the node’s stream is currently being played, blue means this node is the path the user chose to take in the future. In Figure 6.6 we see a graph of a media experience that is being played. We can learn some information about this graph. First, we note that ind stands for individual stream. In the graph we use only display the first letter of ind, namely i. We made this choice to have a readable and compact format. In the figure we see the three different colors: red, blue and
green. Node ind01 is red, this means the frontend finished playing this node. During or after playing ind01, the user made the choice to play node ind07. Node ind07 is green which means it is still being played. Node ind06 is blue, this means the user has already made his choice, even though he is still playing the active stream of node ind07. Once the active stream ind07 has ended, a seamless transition to node ind06 should occur.

Figure 6.6: Graph representation in the frontend with different statuses for some nodes

A user can start the playback of a media experience by using the play button below the ODV Viewer or clicking the root node in the graph. Once the user has initiated playback, media data will be fetched by the backend and forwarded to the frontend. When playing the first stream in a media experience, the user will notice some start-up time delay. This start-up time encompasses the time needed to get enough segment data to reach the minimal buffer time for the initial active stream, the time to create an initial data structure, and the time it takes to start the decoder and start playing the video. When switching to a future stream, the start-up time should be minimal because enough data should be buffered, the data structure is created before the new stream has to be played, and the decoder is started because enough data is present. In the case of future streams, the start-up time hence should only envelop the time to start playing the video.

6.5 Results

In this section we will provide some experimental results related to this use case. For each experiment we will explain what we will test, motivate this decision, provide what dataset we used to test it, explain what the expected results are and what the actual results are. For each experiment we will also make a conclusion based on the gathered results. We will conduct the following experiments: how fast is the “seamless” switching, how much influence has precaching on the performance of the active stream, and how is the available bandwidth distributed between active and future streams. In addition to the experiments listed above we will also conduct the user test “User test: Seamless switching with Path-Based Non-Linear video”.

All the experiments are executed on the following setup: MacBook Pro 13-inch with Retina-display, OSX 10.9.4, 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3 RAM and Intel Iris graphics card. We use the Google Chrome
browser version 39.0.2171.71 (64-bit) as an environment to conduct these tests in. The dataset is located on an external server. The backend will fetch segments using an internet connection with download speed 21 Mbps and upload speed 1.85 Mbps. The backend will be run on the same machine as the frontend, thus communication between the two will be over the laptop’s loopback interface.

6.5.1 How fast is the “seamless” switching?

In this test we determine how long it takes to switch from one stream to the next in our approach. This means we test the time it takes to switch from the active stream to one of the future streams. In this test we can have two scenarios: the user’s choice is made respectively before and after the active stream ends. We define a switch to be seamless if the time it takes to switch from one stream to the other is lower than a certain threshold. For now we use the threshold value of 200 ms \[5\], which represents the time it takes for humans to react to colors and motion changes. In 6.5.2 we will determine a threshold by the means of a user study. In this user study we will also look how good the switching in our solution behaves. In the first scenario we test the time it takes from the moment the active stream ends until the next stream emits the playing event. In the second scenario we test the time it takes from the moment the user chooses a path until the chosen path emits the playing event.

This test will provide us with objective results if our approach supports seamless switching rather introducing stalls in the playback.

We used a dataset that has a range of input files that could potentially give varying results. The dataset consists out of:

- Streams with high bitrate and low resolution
- Streams with high bitrate and high resolution
- Streams with low bitrate and low resolution
- Streams with low bitrate and high resolution

Different bitrates and resolutions will introduce different video file sizes, thus different size segments.

We expect that the time it takes to switch between the streams will be less than the threshold for each of the streams in the dataset. However, we expect that choosing a path after the video ends will take a bit longer to switch because the ViewerManager has less time to prepare the correct stream.

In Table 6.1 we see the results of the experiments for both scenarios. For each stream in the dataset we executed the test 10 times, to see if the results would vary. From the information provided in the table, we can conclude that bitrate and resolution differences do not influence the seamless switching performance. However, there is a slight difference between the two scenario. In scenario one the results vary from 102ms to 106ms, while in scenario two they vary from 116ms to 124ms. The difference between the two scenario can be explained by the fact that the ViewerManager has no time to prepare the correct stream. The ViewerManager waits before switching to the next steam until it receives meta data from the backend. Because the results are all below the threshold, we can conclude that our approach provides seamless switching in both scenarios.
### User test: Seamless switching with Path-Based Non-Linear video

The aim of this user test is to primarily determine if the switching in our solution is perceived as seamless and what the threshold is to mark a switching as seamless. Secondarily we look at how users observe switches between videos and to what extent they accept these switches.

This user test was one out of two tests of an integrated user study involving the same set of participants. The other user test is described in Section 7.5.2. Therefore, some of methodology is the same for both tests, especially the introductory part of the tests. The user study consisted out of three parts: establishing the participant’s profile, rating statements in the context of seamless switching, and rating statements about the high quality viewport approach. First, the participants were given a short introduction about the context in which this user study was conducted. Afterwards, participants were asked to fill in questions to establish their profile. We asked the following questions: “What is your age?”, “How many hours per week do you spend watching videos on a computer?” and “Rate the following statement: I have a basic knowledge on how video streaming works.”. At this point, the introductory part of the user study was finished and users were informed that the next part would talk about the use case Path-Based Non-Linear video. Participants were asked about their knowledge of non-linear video. If the participant lacked the knowledge, missing parts were filled in or explained further. We then explained in what context we use non-linear video in this use case and how this concept can be achieved in practice. In essence videos are played after each other to achieve the non-linear video. We explained the ideal case, where there is no transition delay between subsequent clips in the non-linear video path, and showed this to the participants as a baseline. We then showed a random order of scenarios which each varied the transition time between both videos. For each distinct scenario, participants were then asked to rate the following statements using a 5-point Likert Scale [49]: “I could see a transition between both videos.” and “I find the transition delay between both videos acceptable.”. After all the scenarios were finished, participants were asked about their overall idea of the technique.

Fourteen participants took part in our user study: 13 male and 1 female,
with ages ranging from 18 to 54. Only one participant had no knowledge of how video streaming works, the rest all had a basic knowledge. All participants indicated that they watch videos on a computer ranging from 1 to 15 hours per week (average 7.5 hours).

We conducted this user study with the setup described in Section 6.5. However, no Internet connection was used during this study, which means that the media data was hosted locally.

![Median for user test: transition time](image)

**Figure 6.7**: Median for user test: transition time. The chart displays the median of the rating for visibility and acceptance of the transition delay. The transition delay varied from 50 ms to 250 ms. It also displays the same results for our solution.

In Figure 6.7 we can see a chart that displays the median of the rating for visibility and acceptance of the transition delay. We varied the transition time between 50 ms and 250 ms and determined how our solution relates to the transition delays. First, let us look at the median values for the visibility level. Here we can see that the participants generally started to notice the transition at 150 ms. Transition delays of 200 ms and 250 ms were almost always noticed. Our proposed solution scores very well, the transition time is hardly visible. With one-way ANOVA, we found a significant effect of the transition time on the visibility level (F(5,78)=4.90, p=0.03). In Table 6.2 we can see the post-hoc comparison of all combinations.

When we look at the median value of the acceptance level, we can see that it does not vary a lot. For our solution and transition times 50 ms and 100 ms, the median value is maximal. For the other transition delays, the median value of the acceptance level is still a 4, which is still acceptable. Therefore, we can conclude that in all scenarios the transition time was acceptable. With one-way ANOVA, we found no significant effect of the transition time on the acceptance level (F(5,78)=1.83, p=0.19).
Our solution

Table 6.2: The post-hoc comparison of all transition time combinations. A * is used when a significant effect was found between two transition times. When no significant effect was found, a – is used. We only use the lower part of the table to avoid visual clutter and to not include duplicates.

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 seconds</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100 seconds</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>150 seconds</td>
<td>*</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>200 seconds</td>
<td>*</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>250 seconds</td>
<td>*</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our solution</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

However, we want to know if there is a relation between the visibility of the transition and the acceptance of it. In the chart we can see that when the visibility rating increases (i.e., the transition became more visible), the acceptance rating decreases slightly (i.e., the transition time is less acceptable). We found that the two variables were strongly correlated (Pearson’s r(82)=-0.58, p=6.04e-09).

Users generally accepted the results, even when the transition time increased to a maximum of 250 ms. The participants used the following statements to clarify why they accepted the results: “when streaming videos, stuttering is possible and the transition seemed like a small stutter in the video”, “the transition delay was short enough to not be annoying”, “the transition was consistent, there was no gap between the videos”. Participants noted that the level of acceptance also depends on the video content, they prefer less delay when the video has, for example, an action scene.

When the transition time would increase further, the trend shows us that at some point the resulting experience would become unacceptable for users. However, for all the delays present in this test, users accepted the result. Important factors with the transition time are: (i) the length of the transition time has to be relatively short (i.e., if the transition is visible, it has to look like a small stutter), and (ii) the acceptable length of the transition time should depends on the content of the video.

6.5.3 How much influence has precaching on the performance of the active stream?

We test how the precaching of future streams will influence the performance of the active stream playback. We conduct a test where we measure the performance of our application while precaching and while not precaching. To test this, we use the Google Chrome profiling tool. We use the precache logic that precaches the future streams at the start of the active stream. For each stream in the dataset we measure the performance respectively with and without precaching. To be able to test this, we first measure the time it takes for each stream in the dataset to finish precaching its future streams. Because we manually have to start and stop the Google Chrome profiler, these results are not exact. However, they give a notion of the performance of the application. In this test we also include some results about the perceptual fidelity while con-
ducting the experiments. These results could give extra information about the influence of the precaching. When testing the performance, we look at four work-intensive functions:

- **drawImage**: Draws the current video image on a HTML5 canvas
- **Base64Decoding**: Decodes a base64 encoded string into an uint8 buffer
- **ImageToTexture**: Converts an image to a texture for spherical mapping when using the ODV Viewer described in Section 4.5
- **OnMessage**: Handler for an incoming data message

The test will provide us with results that will pinpoint the computational bottleneck of the application, if there is any. If the influence of the precaching is too high, it can cause the playback of the active stream to stall.

The dataset exists out of three different qualities of the same stream. Segments from a high quality stream consume more available bandwidth to download than those from a low quality stream. We therefore provided a low quality stream, a medium quality stream and a high quality stream.

For the low quality stream we expect that the influence will be minimal because the size of the segments is small and their processing should produce minimal extra work. For the medium quality stream we expect to see an increase in influence but still not so much that it will give stalls in the active stream. In the case of a high quality stream we expect a big influence on the active stream because the frontend has to handle larger segments.

First we determine a time frame for the low quality stream. We use an active stream and two future streams of resolution $2400 \times 800$ and 500 Kbps bitrate. The segments in this case have an average size of 120 Kb. On average, it took 2.5 seconds before all the segments were available in the frontend. For the low quality stream we will therefore use a time frame of 2.5 seconds. Next we determine a time frame for the medium quality stream. We will use both an active and two future streams of resolution $426 \times 240$ and 3540 Kbps bitrate. Segments have an average size of 800 Kb. On average, it took 10 seconds for all the precached segments to be present in the frontend. The determined time frame for the medium quality stream is 10 seconds. As a high quality stream we use an active and two future streams of resolution $2400 \times 800$ and 6000 Kbps bitrate. Segments have an average size of 1400 Kb. On average, it took 18 seconds before all the segments are precached. We will therefore use a time frame of 18 seconds for the high quality stream.

In Table 6.3, we see the results of the conducted experiments. These results are all displayed as percentages of time spent in the function compared to the total time consumed by the test. In the case of the low quality stream without precaching, we spent 0.99% of the total time in the Base64Decoding function. In other words, we spent 1% of 2.5 seconds in this function, which equals 25 milliseconds. We see that even in the low quality case, a lot more time is spent in the functions Base64Decoding and OnMessage. When the frontend receives more messages from the websocket, more messages have to be base64 decoded. Because we draw the first frame of the future streams on the screen, a bit of time is required for this operation. In the results for a medium quality stream we see that the base 64 decoding varies the most between the caching and non-precaching alternatives. We will naturally spend less time in the OnMessage
function because we receive less messages. Just as in the medium quality case, in high quality stream the time spent in the function Base64Decoding varies the most. The perceptual fidelity differs between the low, medium and high quality streams. In the case of low and medium quality streams the playback never stalled, even when precaching future streams. However, in the case of a high quality stream stalls were introduced while precaching. Between 3-6 stalls happen on average when downloading and precaching the high quality streams. These stalls have a duration of a second on average.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wo</td>
<td>w</td>
<td>wo</td>
</tr>
<tr>
<td>drawImage</td>
<td>2.85</td>
<td>3.08</td>
<td>0.58</td>
</tr>
<tr>
<td>Base64Decoding</td>
<td>0.99</td>
<td>2.11</td>
<td>1.84</td>
</tr>
<tr>
<td>ImageToTexture</td>
<td>1.28</td>
<td>1.39</td>
<td>1.20</td>
</tr>
<tr>
<td>OnMessage</td>
<td>0.36</td>
<td>0.75</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table 6.3: Result of performance test in low, medium and high quality. The wo and w stand respectively for active stream without or with precaching. All the results are displayed in percentages of time spent in the function compared to the total time of the test.

From the results in Table 6.3 we can draw some conclusions. We however have to remember that the time frame for each tested quality is different. The time frames are 2.5, 10 and 18 seconds for respectively low, medium and high quality. This time frame makes it possible to see how much work is done in each investigated function during the time it normally takes to precache. We also have to note that the results are not exact because we manually started and stopped the profiler tool. The first conclusion we can make is: for higher resolution streams the application spends more time drawing it on a canvas. A second conclusion: for bigger segment sizes we spend more time decoding the base64 string. We gathered low overall percentages as results. This can be explained by the event-based nature of Javascript and thus our application. When the frontend receives a data message from the backend, it will do the base64 decoding, which can be an intensive task. When we receive no message we only have to draw images on the screen at a certain rate, the rest of the time the application is idle.

6.5.4 How is the bandwidth distributed between active and future streams?

We would like to know how the distribution between active and future streams behaves in a normal playback scenario. There are a few parameters that determine this distribution: when we start to precache, how many streams we precache, the amount of data we precache per stream, and the quality of the precached streams. When we start to precache is determined by the precaching logic; in the current implementation there are two options: precaching in the beginning and during the playback of the active stream. In the current implementation we always precache all future streams, thus how many streams were precache is determined by the amount of future streams of the active stream.
The amount of data we precache per stream is set in the DownloadManager module. It is determined by taking into account the length of the future stream. The quality of the future streams is determined by the QAAdaptation module in the backend. If we choose the best fit quality adaptation logic, it chooses a representation for the future streams taking into account the number of other future streams and the available bandwidth. The quality adaptation logic chooses representations for the future streams that can be played with 40% of the available bandwidth. We do some tests with different kinds of configurations and explain the obtained results. We do not test all permutations of configurations but rather select a few that could give meaningful information. To actually get results about this distribution, we use a bandwidth monitor provided by EDM. The bandwidth monitor displays the amount of bandwidth used by active stream and future streams.

We want to measure the distribution to get a better view of the backend’s bandwidth consumption behavior in a normal playback scenario.

The dataset contains three media experiences:

- A single low quality representation media experience and two children
- A single medium quality representation media experience and two children
- A media experience with nine representations ranging from low to high and two children
- A media experience with nine representations ranging from low to high and four children

For the single low quality representation we expect to see the download of the segments to end shortly after initiating. The medium quality representation will take longer to download. When we use the best fit quality adaptation logic together with the nine representations media experience, the quality adaptation should choose representations that can be handled by the currently available bandwidth. When we have more children, the chosen representation for each future stream should be lower when bandwidth is steady. In all cases we will download segments as fast as possible because a scheduler is not yet implemented in the framework.

In Table 6.4 we display all the configuration permutations used within this test.

![Figure 6.8: Test1: a single low quality representation with the always lowest quality and precaching at the beginning.](image)

We now take a look at the results of all these experiments. For each test two stacked graphs are displayed. The left graph always shows the consumed
Table 6.4: All configuration permutations tested during this test. We use abbreviations to conserve space inside the table. B stands for precaching at the beginning. P stands for parameteric precaching. AL, AH and BF respectively stand for always lowest quality adaptation, always highest quality adaptation and best fit quality adaptation. The stream quality can be low, med (medium) or varying (/) thus having multiple possible qualities. Available bandwidth is expressed in Mbps.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representations</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Stream quality</td>
<td>low</td>
<td>low</td>
<td>med</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>Future streams</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Precaching logics</td>
<td>B</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>P</td>
<td>B</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>QAAdaptation logics</td>
<td>AL</td>
<td>AL</td>
<td>AL</td>
<td>AH</td>
<td>AH</td>
<td>BF</td>
<td>BF</td>
<td>BF</td>
<td>BF</td>
</tr>
<tr>
<td>Available bandwidth</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>9</td>
<td>18</td>
</tr>
</tbody>
</table>

Figure 6.9: Test2: a single low quality representation with always lowest quality and parametric precaching.

Figure 6.10: Test3: a single medium quality representation with always lowest quality and precaching at the beginning.

Figure 6.11: Test4: a nine representations media experience with always highest and precaching at the beginning.

bandwidth divided into active nodes and precached nodes. The right graph shows the consumed bandwidth for precached nodes.

In Figure 6.8 we see a lot of segments being downloaded at the start. The
downloading process only takes 5 seconds. Precaching uses double the amount of bandwidth compared to the active stream.

In Figure 6.12 we see a clear example of parametric precaching. First the active stream was downloaded and afterwards, when the playback reaches 75% of the total duration, the future streams.

In Figure 6.13 we see that the downloading process takes a bit longer. It takes about 16 seconds to download all the necessary segments. This can be explained because the quality of these streams is higher than in the previous examples.

In Figure 6.14 a lot of data is being downloaded at the start. This is greatly influenced by the precaching. Once the precaching ends, we use more available bandwidth for the active stream. This is not caused by the quality adaptation logic but because of the basic implementation of the SegmentDownloadRateEnforcer.

In Figure 6.15 we see two big download iterations. The first will download the active stream at about 12 Mbps. The second will precache both future streams. We generally do not precache a full stream, the data being precached is only a part of the streams. To second iteration is almost as big as the first.
one because we precache two streams. Parametric precaching together with the always highest adaptation logic in this case leads to not complete precaching before the active stream ends.

In Figure 6.13 we use a more intelligent quality adaptation to download the best quality fit for the available bandwidth. We see that compared to test4, the download process takes less time. This is caused by downloading lower representations that can actually be handled by the current network conditions. The best fit quality adaptation logic chose representation 8 for the active stream and 6 for both future streams.

In Figure 6.14 we can see two iterations of downloads similar to test5. If we would use the same quality for the future streams as for the active stream, the precaching would take longer and would probably not complete before the end of the active stream. The best fit quality adaptation chose the same representations as in test6, representation 8 and 6.

In Figure 6.15 a similar downloading pattern as test7 occurs. The overflow on the bandwidth axis is caused by artificially setting the available bandwidth to 9 Mbps in the frontend, while 18 Mbps is actually available on the network. In this case the best fit chose for representation 7 for the active stream and 4 for the future streams.

In Figure 6.16 we see the precaching of four future streams. Each stream will use less available bandwidth compared to test7. This is caused by the choice of a lower representation 4 for the future streams. For the active stream we will still use representation 8 because nothing changed to the scenario of the active stream compared to test7.

These results give a more clear view of the bandwidth distribution. As expected the best fit quality adaptation logic chooses a representation that can be handled by the current network conditions. When we preache at the start, a lot of data is downloaded at once. When we use parametric precaching, the

Figure 6.15: Test8: a nine representations media experience with best fit and parametric precaching at 9 Mbps.

Figure 6.16: Test9: a nine representations media experience with four children with best fit and parametric precaching at 18 Mbps.
downloading is split into two iterations. When we have multiple children, the available bandwidth for these children will be divided equally.

6.6 Conclusion

In this use case we proposed a solution that enables users to “walk” a virtual tour. Users can personalize this tour through the concept of non-linear video. This non-linear video exists out of multiple sequences of video footage. These sequences are used to gradually build the user’s virtual tour. In this use case we describe a solution to avoid stalls between these sequences.

Our approach relies on a combination of functionalities in the frontend and the backend. To be able to send non-linear videos, we describe the mutual relation of MPDs inside the media experience description. By carefully monitoring the relation between active streams and future streams or solution is able to provide media data for these future streams in time. The fact that this data is present in the frontend means that the application seamlessly switches between streams. Because this kind of experience is a different experience compared to normal video streaming, we provide some visual support in the form of a graph. This graph not only shows users their current location inside the tour but also lets them interact with it.

By conducting experiments, we obtained objective results that our solution provides seamless switching between streams. Results from the user study indicate that even if a small transition is visible between streams, users generally accept the result. They refer to it as a “small stutter” in the video. Participants indicated that the acceptance level of the transition delay depends on the content of the video. For example, they want lower transition time between video in an action scene. The performance of our web-based solution is high in the case of low and medium quality video. Some performance issues arise when streaming high quality video while precaching the same quality video. Results show that deciding between precaching logics can be very important. Precaching at the beginning consumes a lot of bandwidth at the start while parametric precaching leads consuming bandwidth in two iterations. The more advanced best fit quality adaptation logic makes sure the backend fetches segments that can actually be streamed under the current network conditions.

In this use case we give users a different experience compared to regular video streaming. As with all new techniques, users have to learn how to use it. Our solution provides the user with visual support, in the form of a graph, to lower the learning curve.

Compared to the related work we provided a more readable and efficient way of describing a non-linear video. The media experience file enabled us to maintain oversight while playing the non-linear video. Having separate prefetching policies would enable our solution to maximize the quality of future streams.

Our solution is currently not able to efficiently stream and precache high quality videos. Improving the performance of some crucial functions, would perhaps enable us to efficiently stream and precache this high quality. Currently the base64 decoding of incoming data is slow. When data messages are received, we first decode them before doing anything else. Also when data is not needed right away, we still decode it. Therefore a solution to this problem might be a scheduler for decoding data messages. Performances issues in a video player are
noticeable when the video starts to stall. Therefore an other solution could be to give priority to some functionality. We could give high priority to drawing the active stream on the screen and a lower priority to decoding future stream data messages.

As future work, this solution could be enhanced to allow streaming high quality videos. Visual support for the user could be enlarged by showing a map of the virtual tour. In this map the current location within the tour could be marked. In our solution we created a web-based solution. This technique can also be used in other areas for instance VR.

In the context of a virtual tour, our solution provides all the functionality I would expect. Being able to seamlessly transition between streams makes the experience very fluent. It is important that stalls are not frequent or non-existent because these would ruin the experience. Having visual support lowers the learning curve in a scenario like this. Having a solution that works perfectly for each quality of streaming while still precaching would be ideal. The proposed solution however provides enough added value in the context of a virtual tour.
Chapter 7

Tiled ODV distribution with viewport resolution optimizations

7.1 Introduction

In this thesis, we often use omni-directional videos as media material to test our use cases. Omni-directional videos generally have a higher resolution compared to regular videos. This is explained by a simple fact: the field of view of an ODV is bigger compared to a normal video, thus existing out of more pixels. We generally only view a part of the ODV at all times, this is called the viewport. The video footage within this viewport typically has a resolution that is comparable to that of a classic video clip. To view video content outside the viewport, a user has to interact with the ODV Viewer, as described in Section 4.5, to look around. To account for additional video content outside the viewport, the ODV file will be larger compared to regular videos.

To clarify the file size of an ODV, we use an example with two videos: an ODV and a normal video which contains a spatial segment of the ODV. In the ODV, we can view the same video content as in the regular video clip. A user can have the same video experience in both videos when not interacting with the ODV. In Section 3.1, we explained that users have high expectations regarding video experience. An important part of the video experience is the visual quality of the video. This means users expect videos to be provided in a high quality. High quality videos are defined by a high resolution and high bitrates. When we use this knowledge in the example of the two videos we can conclude: to have the same quality for the video as the viewport of the ODV, the resolution of the ODV must be bigger. In Figure 7.1 an ODV is displayed with a red box highlighting a possible viewport. In Figure 7.2 we see this viewport in more detail. However, this video could just as well have been a separate video file. When we compare both figures, we can understand that the resolution of an ODV will be a lot bigger than that of a normal video.

To give a concrete idea of how big such a file is, we again use an example. Because we can correspond the viewport of an ODV with a normal video file,
Figure 7.1: One frame of an ODV where we highlight the part of Figure 7.2.

Figure 7.2: The same information as highlighted by the box in Figure 7.1. This could be the viewport of an ODV Viewer or a separate video file.

the viewport in this example consists out of 1280 × 720 pixels. If we want to be able to look left and right in this ODV over a distance of half a viewport, there should be sufficient video data available to make this possible. In terms of pixels, this means we should have 640 × 720 pixels to the left and right of the viewport. This results in an ODV file of resolution 2560 × 720, which contains double the amount of pixels of the original viewport. In this example we were only able to look left and right over a short distance. Providing more freedom to look left and right will result in much higher resolutions. However, remember that we increase the resolution to keep the video quality comparable to the ones users are accustomed with when interacting with non-ODV content.

Streaming large video files over the internet can take some time. In essence ODVs are just large video files, and thus they suffer from the same time constraints. The big file size is a problem to transfer over the internet. The big file sizes are caused by resolution and bitrate of the video file. To lower the file size, we have to decrease those parameters. However, when we want to keep the
overall quality high, we should look for other solutions. In this use case we will look at a possible solution to tackle the problem of sending a lot of video data over the network in the case of ODVs.

The idea behind this use case is that a user at all times only watches a part of the ODV, the so-called viewport. To view video data outside of the original viewport, a user has to interact with the ODV Viewer. In this use case we provide a solution that exploits this behavior. To maximize the quality of the video experience, we should provide the viewport in high quality. To save bandwidth, we send the video data outside the viewport in a lower quality. The downside to this approach occurs when users start to pan. The perceptual fidelity of the new viewport will be low. When a pan occurs we should try and provide the new viewport in a higher quality as soon as possible.

Because we want to be able to send parts of the ODV in higher quality than the rest, we will spatially divide the ODV in tiles. Tiling a video will result in a number of tiles which each hold different parts of the original video. Because these tiles relate to the original video, it is important to remember their mutual spatial relation. For example, we could divide an ODV in four tiles: top left, top right, bottom left and bottom right. All these tiles contain a part of the ODV. The tiles can then be used to play the content of the ODV when we position them in the right order. When playing a tiled video, we generally do not want users to notice the separate tiles. Thus it is important that the playback of all tiles is synchronized and no edges between tiles are introduced. In Figure 7.3 an ODV is divided into four tiles. This figure clearly shows the importance of the spatial relation between tiles. To get a meaningful end result we should respect the position of each of these tiles.

To adapt the viewport to high quality, we have to determine what tiles are displayed within this viewport. These tiles should then be requested in high quality, while requesting the rest in a lower quality. Because the tiles only in certain cases directly map to the viewport, some parts outside the viewport will also be sent in high quality. When tiles are relatively big, more high quality data that falls outside of the viewport will typically be sent. Dividing a video into more tiles will limit the extra amount of high quality data we send. However, when we use a lot of tiles, syncing them will become harder. Syncing two or four tiles is doable, but syncing a lot of tiles can be a hassle.

When combining tiles to display the origin video, the perfect scenario would be 100% synchronization all the time. When only one tile is a few frames
delayed, the result is already noticeable.

In Section 7.2 we will look at related work about tiling approaches. In Section 7.3 we will talk about our approach to create a solution that is able to stream a high quality viewport. For each use case a specific frontend is built; in Section 7.4 we will describe this application in depth.

### 7.2 Related Work

Video resolutions have been evolving during the years. Not too long ago, HD videos were widely used. Recently, there has been a transition from HD videos to resolutions of 4K. In the near future a transition to 8K resolution is believed to happen. Because of this a lot of research is done in the field of tiled video, to support the streaming of high resolution video. We can use the same research for the purpose of our use case. In our case the resolution is also high but is caused by playing ODV content.

Wang et al. have described a perceptual quality assessment for mixing tile resolution in tiled video. In this paper, a proposition is made to use a fixed number of tiles to encode and decode video instead of fixing tile size. Each of the tiles will be provided in different resolutions. In Figure 7.4 we can see the process to create such a tiled video. This tiling approach stands in contrast to other tiling approaches. Other approaches create more tiles for higher resolution videos and create a low number of tiles for lower resolution videos. In such an approach a variable number of tiles will be created, thus the client does not know beforehand how many tiles it has to decode. In the approach with a fixed number of tiles, the decoder knows how many tiles it has to decode each time. Because different tiles can have different resolutions, tiles should be scaled up or down depending on the requested resolution of the entire video. The mixed resolutions tiling scheme is proposed because it has two advantages: the approach scales well for multicast transmission and can reduce bandwidth consumption without reducing the perceived video quality. In their case they can provide regions that are requested by a lot of people in a higher quality compared to regions that are only requested by some people.

![Figure 7.4: The process to create a tiled video, where we have a fixed number of tiles and each tile can have different resolutions](image)

The psychophysical study conducted by Wang et al. uses the method of limits to measure two thresholds: just noticeable difference (JND) and just unacceptable difference (JUD). To conduct the tests tiled videos are used where the tile resolutions are randomly chosen from two resolution levels with equal probability. To obtain a result for different kinds of video, three types of videos are used: dense motion, medium motion and low motion. The most notable
results from this study are:

- when tiles from a 1920 × 1080 and a 1600 × 900 stream are mixed together, users hardly notice this; even when noticed, the difference is generally accepted
- tiles from a 1920 × 1080, a 1600 × 900 and a 1280 × 720 stream can be mixed where more than 85% of the test subjects accept this configuration
- high motion video is more sensitive to resolution mixing
- the quality degradation from mixing tiles of resolution ratio 80 × 45 is slightly less obvious than from mixing tiles of ratio 16 × 9.

Devloo et al. have proposed a design and evaluation of tile selection algorithms for tiled HTTP adaptive streaming [20]. To maximize the quality and resolution of a video and counteract the bandwidth problem of high resolution videos, the authors use tiled HTTP adaptive streaming (TAS). TAS is a technique that attempts to minimize bandwidth costs by subdividing a video segment both spatially and temporally and encoding each segment in multiple quality levels. Spatial subdivision is done by dividing the video in multiple tiles. Temporal subdivision is used to be able to provide adaptive streaming, similar to MPEG DASH. The tile selection algorithm determines which tiles have to be downloaded in what resolution. Devloo et al. describe three algorithms used for different TAS techniques: scaled down video, cropped video and pannable (and zoomable) video. In Figure 7.5 we can see the evaluated use cases for TAS. In contrast to [73], tiling a video will always result in tiles of the same resolution. A set of tiles at a resolution and quality is called a representation layer. The lowest representation layer will only use one tile, while higher layers use more and more tiles. Their work is built upon a prototype TAS client-server infrastructure, developed in the context of the European FP7 FascinatE project. This prototype includes a basic tile selection algorithm which will be used as a basis for comparison.

The proposed algorithm for scaled down video is fairly simple. When scaling down the resolution of the video, high quality details are not visible anymore. This means that when scaling down, we do not need this information, thus do not have to send tiles in this high quality. The algorithm determines what the highest quality is that is required by the client’s screen resolution and that can be streamed through the network. The proposed cropped video algorithm selects the highest possible selection of tiles while taking into account the region of interest (ROI). Because it is not possible to pan in a cropped video, only the ROI has to be downloaded. The proposed pannable and zoomable video algorithm is similar to the cropped video algorithm in that it will send the highest possible selection of tiles for the ROI. However, in contrast to cropped video, users are able to pan. The algorithm will thus send video data for tiles outside the ROI. The choice is made to send this data in a lower quality. Tiles located further away from the ROI will be assigned lower qualities than tiles near the ROI. Because zooming is also an option, the remaining bandwidth is used to increase the quality of the ROI.

The basic tile selection algorithm always requests two streams for every tile: a fallback segment with lowest quality at high priority and a segment of the
chosen quality at normal priority. The fallback segment is meant as a backup when the other segment cannot be downloaded in time. In the case of scaled down video, the algorithm will fall back to the lowest quality when not enough bandwidth is available. When a bandwidth drop occurs in the case of cropped video, the algorithm will be unable to download all the tiles and thus use the lowest quality for some tiles. In the case of pannable and zoomable video, panning and zooming will cause the algorithm to use the fallback layer for tiles that were outside the ROI when experiencing severe bandwidth constraints.

The algorithms proposed by Devloo et al. try to improve the overall quality of the video compared to the basic tile selection algorithm. In the case of scaled down video and cropped video the algorithm will not use the fallback layer but will calculate which quality can be downloaded with the current network conditions for a set of tiles. In case of pannable and zoomable video, the proposed algorithm has higher quality tiles outside the ROI and thus will not fall back to the lowest quality. As explained above the quality for tiles outside the ROI is dependent on the distance of the tile to the ROI. The authors do not mention how this distance is calculated.

7.3 Approach

In our approach we aim to reduce bandwidth consumption of an ODV or other high resolution videos by providing only a high quality for the parts the users actually see. In an ODV, the part a user can see is the viewport. This viewport is a synonym for the ROI in paper [20]. However, in contrast to this paper, we will use a static number of tiles, which is similar to the approach of [20]. Figure 7.6 explains the tiling approach we use in the remainder of this use case. In this figure we divided the video into four tiles. Each tile has the same size.
and has different qualities. In contrast to the number of tiles has no effect on the actual quality of the video stream.

Tiling a video is part of making the dataset; it is a preprocessing step. To divide a video into tiles we can for example use the tool FFMPEG. This tool can divide a video into multiple tiles and encode different qualities for each tile. Because in this use case we also use MPEG-DASH, the videos should be described by an MPD. Each tile can be looked at as a separate video, thus we can describe them in a separate MPD.

![Figure 7.6: Our tiling approach in the case of four tiles. Each tile has the same size and can have different qualities.]

To be able to stream media data with the implementation framework, we need to create a media experience file from the generated MPDs. To incorporate tiling into the media experience, we can use the syntax defined by the media experience file. We can define specialized nodes that describe a tiled video, we call them groups. A group will have three important fields: id, isGroup and parItems. The id should be unique, because multiple groups can be defined in the media experience. Because a group is created as any other node, we included a field isGroup. The attribute parItems contains a list of node ids which are contained in this group. Each node that is part of the group should provide their position in the ordering in the field activeOutWidget.

In Listing 7.1 an example is given for a tiled video consisting of four tiles. It describes the same structure as shown in Figure 7.6. Because the grouping of tiles is a preset configuration, it is important to keep a fixed number of tiles. Using a variable number of tiles does not map well onto the structure of the media experience description. To be able to provide multiple qualities, each MPD should contain multiple representations for its tile. When this is the case, the adaptive streaming functionality of the implementation framework can be reused to stream high quality for tiles inside the viewport and a lower quality for tiles outside of the viewport.

```json
1 {  
2     "nodes": [  
3         {  
4             "id": "grp01",  
5             "isRoot": true,  
6             "isGroup": true,  
7             ...  
8         }, ...
9     ]
10 }
```
In the media experience file, a group contains information about which streams should be displayed. The position of each stream within the group is defined by the attribute `activeOutWidget`. The value for the `activeOutWidget` can range from 0 to the number of tiles in the group. Each value describes a location within the group in relation to the total number of tiles in the group. When groups are defined, the backend fetches segments for each stream in this group at an equal rate. At the frontend, all these segments are received in separate messages. The spatial order of the streams is provided to the frontend after choosing the media experience. With this information, the frontend is able to recreate the full video image from all the separate streams. Each stream is stored inside a viewer. To have a notion of a group in the frontend, we created a class called `ViewerGroup`. This `ViewerGroup` contains viewers related to the same group and their spatial information. To be able to display the video on the screen, we have to draw each viewer of the group in its defined position. To recreate the integral image we use a HTML5 canvas. Every time a video frame changes, we draw the images from all the viewers onto the canvas at the correct position. We created an algorithm that can draw a group onto the canvas, with known spatial information. The algorithm to position the streams currently only works for an equally divided grid, for example $2 \times 2$, $3 \times 3$ and $4 \times 4$. In
Listing 7.2 we see the algorithm used to draw a group on a canvas. First the algorithm determines how many viewers should be displayed next to and below each other. In the case of for example $4 \times 4$ we should draw four viewers next and below each other. When we know this amount, together with the size of the canvas and the fact that all tiles have the same size, we can determine how many pixels each viewer will contribute to the canvas.

Because tiles have a different aspect ratio than $1 : 1$, the horizontal tile size will differ from the vertical tile size. Afterwards the algorithm determines the top left coordinate of the position where each viewer should be drawn. When we know the top left position and the size, we can use this information to draw the viewer on the correct position.

```
function drawToCanvas(canvas, group) {
    var size = group.length; // # of viewers inside the group
    var length = Math.sqrt(size); // Determine # of viewers
    // next to and below each other
    var hor = canvas.width / length; // Horizontal size of the tile
    var vert = canvas.height / length; // Vertical size of the tile
    var groupIndex = 0;
    var draw = canvas.getContext("2d");
    draw.clearRect(0, 0, canvas.width, canvas.height); // Clear the canvas
    for (var v = 0; v < length; v++) {
        for (var h = 0; h < length; h++) {
            var beginX = h * hor;
            var beginY = v * vert;
            groupIndex = h + v * length;
            var view = group.viewers[groupIndex];
            draw.drawImage(view.video, beginX, beginY, hor, vert);
        }
    }
}
```

Listing 7.2: Drawing algorithm for a ViewerGroup

Tiles are being drawn in a certain position to “fake” the notion of having one video. To have an ideal solution users should not be able to see the separate tiles. This means all the tiles have to be in sync when playing. In a webbrowser environment syncing between multiple videos can be hard. A HTML5 video contains a variable currentTime, which tells the current playback time of the video. We can use this together with the video controls play and pause to stall tiles when synchronization problems occur. This method is rather inaccurate to sync more than two videos. The holy grail of synchronizing videos in a webbrowser is defined by the HTML5 standard. The HTML5 standard defines video/audio property mediaGroup to allow two or more audio and video elements to be kept synchronized. The HTML5 mediaGroup attribute can be used in combination with the HTML5 MediaController. A MediaController is an object that coordinates the playback of multiple media elements. By using this combination, we
can create a MediaController for a mediaGroup and use the controller to play all the videos in the media group. This kind of setup can be used to play and sync multiple videos. However, browser implementations for this feature are minimal or non-existent. In Listing 7.3 we see a simple example on how to use a mediaGroup and mediaController to play two videos synchronously.

```javascript
1 var video1 = document.getElementById("myvideo1");
2 var video2 = document.getElementById("myvideo2");
3 var mediaController = new MediaController();
4 video1.mediaGroup="media";
5 video2.mediaGroup="media";
6 video1.controller = mediaController;
7 video2.controller = mediaController;
8 mediaController.play();
```

Listing 7.3: Simple example of how to use a mediaGroup and MediaController to synchronize playback of two videos

Google Chrome had an implementation for the MediaController but un-shipped this feature because in their implementation the MediaControllers functionality was minimal. Because the MediaController was very rarely used and not supported in Firefox or IE, the risk to unship it seemed low [42]. The offered functionality was not much more than playing multiple videos at the same time. While waiting for a more functional implementation of this part of the HTML5 standard, we use this temporary “solution”. For each Viewer in a ViewerGroup we initiate playback at the same time. When we wait until enough data is buffered for each viewer, each viewer is able to play without interrupts, so the viewers will stay more or less in sync. By using this method, we synchronize the start point of the playback.

At this point we are able to play a tiled video with our current DASH implementation framework. Now we can extend this functionality, to be able to receive high quality video for the viewport. To be able to stream the viewport in high quality, we first must determine which tiles are inside the viewport. A viewport is described by a top left coordinate and its width and height. We created an algorithm that will check whether a viewer is contained in the viewport. Because we also made the algorithm to display each viewer on the canvas, we can reuse this structure to make this checking algorithm. The algorithm iterates over each viewer the same way as in Listing 7.2. For each viewer the algorithm checks if it overlaps with the viewport. In Listing 7.4 the test to determine if the viewer overlaps with the viewport is shown. We will use the coordinates of the left top and right bottom of each viewer and compare it to the viewport’s coordinates. For the viewer these coordinates are stored in respectively beginX, beginY, endX and endY. For the viewport the algorithm stores them in viewport.startX, viewport startY, viewEndX and viewEndY. The algorithm in essence executes two tests:

1. are the end coordinates of the viewer larger than the begin coordinates of the viewport
2. are the begin coordinates of the viewer smaller or equal to the end coordinates of the viewport.

Test1 ensures that the beginning of the viewport is located left of the end of each viewer. Test2 ensures that the viewport ends to the right of the beginning
of each viewer. When both tests succeed part of the viewport overlaps with the
viewer. Because this use case will be tested with ODV, we should account for
video being circular. ODV is mapped onto a sphere, which means that we can
turn 360 degree and always see a part of the video. This also means that
the viewport variables viewEndX and viewEndY can extend beyond the canvas size.
For practical use this means that a viewport can begin at the right boundary of a
video and continue on the left boundary. An example of such a scenario is shown
in Figure 7.7 where we highlight the viewport shown in Figure 7.8 with a red
box. Because of the spherical mapping respectively the left and right side of the
viewport are located at the right and left boundary of the ODV frame. In this
case when we add the dimensions of the canvas to the equation, we “fake” the
same circular behavior in the positions of the viewers. When iterating over all
viewers, this algorithm will select each viewer that overlaps with the viewport.
In Figure 7.9 we shows the relation between viewer and viewport coordinates.
We can see that the end coordinates of the viewport can fall outside of the
canvas. When this is the case, the algorithm checks the circular tests.

```
1 if ( ( endX > viewport.startX && // viewport inside canvas
2     beginX <= viewEndX || // viewport inside canvas
3     endX + canvas.width > viewport.startX && // circular
4     beginX + canvas.width <= viewEndX ) // circular
5     &&
6     ( endY > viewport.startY && // viewport inside canvas
7     beginY <= viewEndY || // viewport inside canvas
8     endY + canvas.height > viewport.startY && // circular
9     beginY + canvas.height <= viewEndY ) ) // circular
10 {
11     console.log("This viewer is part of the viewport");
12 }
```

Listing 7.4: Test to check if the current viewer overlaps with the viewport. This
test should be done while iterating over all viewers drawn on a canvas.

**Figure 7.7:** An entire ODV frame where we highlight the viewport of Figure 7.8.
The viewport is located on the right and left boundaries of these ODV frame.

For each viewer that overlaps with the viewport, the algorithm stores the id
in a list. After all viewers have been checked, the frontend sends this list to the
backend. In the backend a quality adaptation logic is created that chooses the
highest representation for each stream that appears in the list. Streams that
are not in the list, are not part of the viewport, thus the quality adaptation
chooses the lowest representation for them. The quality adaptation logic is
still very basic, but sufficient to show the concept of having a high quality
viewport. There surely is room for improvement with the quality adaptation.
logic, for example we could choose a higher representation than the lowest one for streams outside the viewport when there is enough bandwidth available.

Quality changes to the viewport will not be instantaneous, because each viewer will have some segments buffered. These segments were already requested in a lower quality before the viewport changed. We first have to play these segments before we reach the newly requested high quality segments. Using this method we thus ensure that new segments are sent in a higher quality for the viewport.

Currently data is sent to the frontend as soon as possible, the buffer for each viewer is thus filled as soon as possible. This does not map well to the high quality viewport approach because it could be possible that a lot of data is already buffered at the time the viewport changes. In this case it will take a long time before the viewport’s contents is upgraded to high quality. In the future this will be solved within the implementation framework by modifying the behavior of the SegmentDownloadRateEnforcer, so that it determines when segment fetching should be done rather than always fetching as soon as possible.

### 7.4 Application

To play a tiled video in our frontend, it is sufficient to choose a media experience that contains a group of streams. The frontend will automatically draw these streams in the correct order. When a user wants to have the high quality viewport functionality, he has to select the correct quality adaptation logic. The tiling approach can still be played with other quality adaptation logics but will not give the same result as the high quality viewport logic.

When playback is initiated, we will start receiving media data for each stream. The ViewerGroup will make sure each viewer has buffered enough data before initiating playback. It will wait until the minimal buffer time for each viewer has been reached. The minimal buffer time for the ViewerGroup is
sent by the backend when choosing the media experience. The backend will only
send one minimal buffer time value, even though each tile can have a different
minimal buffer time within its MPD. The backend chooses the minimal buffer
time of the first tile in the group. In the future, the implementation framework
could be improved to send the minimal buffer time for each stream if this is re-
quired for the specific use case. In our case however, one value for the minimal
buffer time is sufficient. Having one value for the minimal buffer time means
that each viewer cannot play before it has buffered more than the predefined
level. Once the buffer level has been reached for each viewer, the playback is
initiated for all viewers at the same time.

If we use the high quality viewport approach, the viewport should be shown
in high quality. When a user pans left or right, he will temporarily see the
lower quality. After all the buffered data is played that was received before the
panning interaction occurred, we will see the new viewport in high quality. In
Figure 7.10 we can see an example of the process that will happen.

When the user interacts with the ODV Viewer, it is possible that the view-
port changes. Zooming and panning will change the viewport. The application
will recheck what tiles overlap with the viewport if the viewport changes. The
user will not notice these checks, but will only notice when the viewport or
a part of the viewport is shown in lower quality. After some time, this lower
quality will be replaced with the highest quality. When all following segments
are already buffered in a low quality, the frontend will not request any new
segments and thus the frontend will be unable to switch to a high quality for
the viewport.

Our application is able to play a tiled video with tiles more or less in sync.
To address bandwidth problems, we added functionality to stream only the
viewport in high quality. In Section 7.5 we will determine two things: is the
high quality viewport approach actually more bandwidth efficient than sending
an entire high quality video, and in what way are users bothered by lower quality
tiles after interacting with the ODV Viewer.
Figure 7.10: A panning process in the high quality viewport use case.
7.5 Results

In this section we want to determine how good our provided solution is compared to sending the entire high resolution video over the network. To determine this we ask the following questions:

- how does the bandwidth consumption of our solution relate to that of an approach where the video is integrally transmitted in high quality
- how do users rate the use of our solution

The first experiment provides objective results about the bandwidth distribution. We provide bandwidth consumption results for different scenarios related to this use case. This experiment is executed on the following setup: MacBook Pro 13-inch with Retina-display, OSX 10.9.4, 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3 RAM and Intel Iris graphics card. We use the Google Chrome browser version 39.0.2171.71 (64-bit) as an environment to conduct this test in.

The second experiment describes a user study to determine if our solution is generally accepted by users.

7.5.1 How does the bandwidth consumption of our solution relate to that of an approach where the video is integrally transmitted in high quality?

We want to determine the difference in bandwidth consumption between certain scenarios. For each test we use the bandwidth consumption by streaming the high quality as a baseline. We compare this baseline with different tiling scenarios. To see how much overhead the tiling approach introduces, we first measure the bandwidth consumption of a scenario where all the tiles are streamed in high quality. In a high quality use case, multiple tiles can overlap with the viewport. To account for a real life scenario, we create multiple tests with different numbers of tiles inside the viewport. We also measure the difference between sending tiles outside the viewport in low quality and in medium quality.

These tests will clarify whether our approach can solve the bandwidth problems introduced by streaming high resolution videos. Testing these scenarios gives a view where our approach is better and where it can be improved.

The dataset in these tests consists out of one video and the tiling of this video: a $2400 \times 800$ video, tiled in a $4 \times 4$ grid. The dataset is located on an external server. The backend will fetch segments using an internet connection with download speed 21 Mbps and upload speed 1.85 Mbps. We expect to see a lower bandwidth consumption when only tiles within the viewport are streamed in high quality and the rest in a lower quality. Sending all tiles in high quality will still consume a lot of available bandwidth similar to the baseline. When only one tile is part of the viewport, we expect to see the lowest amount of bandwidth consumption compared to having more tiles overlap with the viewport.

To display the consumed bandwidth of one or multiple streams, we will use stacked graphs. We only tested each scenario once, thus the actual results can vary when conducting the same test again. In Figure [7.11] we see the consumed bandwidth for the baseline test. In this test we used a stream of 6000 Kbps.
Figure 7.11: Baseline for the following tests. Consumed bandwidth for a high quality stream of bitrate 6000 Kbps.

Figure 7.12: Legend used for the 16 tiles. These colors are used for coloring the stacked portions in the following tests.

Downloading the full stream takes 36 seconds. The throughput will never surpass a limit of 14 Mbps. Figure 7.12 shows the legend for all following tests. By showing it here, we do not clutter the results of these tests. In Figure 7.13 we streamed every tile in high quality. Stated differently, in this test every tile was part of the viewport. The download process took 20 seconds. We can see that the throughput is higher in the case of tiling compared to streaming the video file integrally at high quality. The limit of 14 Mbps of the baseline was easily surpassed. The actual consumed bandwidth stays about the same in both cases. However, due to the parallelism of fetching multiple tiles, we enjoy a higher throughput. From this point forward, we tested scenarios where only a few tiles overlap with the viewport. For each of these scenarios we tested sending the tiles outside the viewport in low quality (a) and medium quality (b). In Figure 7.14 we see the results of a scenario where the viewport was integrally encompassed by a single tile. Downloading will be very fast in both the low quality as with the medium quality compared with the baseline. However, in the case of low quality outside the viewport, downloading completes faster. We see that the one stream in high quality uses more bandwidth than the rest. In Figure 7.15 we see the results of a scenario with two tiles inside the viewport. We can see

1With throughput we mean the rate of successful message delivery over a communication channel measured in bits per second.
that we consumed more available bandwidth to download the streams. The
download time is about the same as in Figure 7.14 which can be explained by
the lack of a DownloadSegmentRateEnforcer. In figures 7.16 and 7.17 we used
scenarios with respectively four and eight tiles inside the viewport. We see the
same results as in previous tests where we consume more available bandwidth
and need longer download times when more high quality segments have to be
Figure 7.17: Consumed bandwidth for eight tiles in high quality.

downloaded.

From these results we can conclude that the tiling approach has some benefits. Fetching multiple streams at the same time gives the ability to reach a higher throughput, thus using the available bandwidth in a more beneficial way. This behavior can be explained because multiple streams can be fetched in parallel. This effect is enlarged because the backend has no decent functionality for the SegmentDownloadRateEnforcer. The DownloadManager only downloads the next segment after it finished handling the previous segment. Having more tiles inside the viewport causes to download more data compared to having only one tile inside the viewport. Normally when we have to download more data, the downloading takes longer. However, when the link is utilized better, thus having a higher throughput, downloading more data does not mean the download will take longer compared to the case with lower throughput. Streaming the tiles outside the viewport in low quality or in medium quality, is dependent on preference. In cases where enough bandwidth is available, downloading these tiles in medium resolution causes no harm. When available bandwidth is limited, download times and video quality should be compared with each other.

7.5.2 User test: How do users rate the use of our solution?

In this user test we tried to determine how users rate our solution. We identified three parameters that could influence a user’s decision whether or not to use our solution:

- the switching time, the time it takes to improve the low quality tiles outside the viewport to a higher quality
- the quality of the tiles outside of the viewport
- the amount of lower quality pixels present in the viewport after a pan.

By adjusting these parameters, we can determine when a user would or would not use our solution.

This user test was one out of two tests of an integrated user study. The other user test has already been described in Section 6.5.2. In this section we only describe the third part of the user study. First, participants were given the explanation that this use case uses a technique to stream an as high as possible video under limited bandwidth conditions. The observer then explained that the ideal situation would be to stream a high quality video without bandwidth constraints. Participants were then shown this ideal situation. Afterwards, two techniques were shown in a random order. One technique displayed a video
with an overall lower quality, while the other showed our solution. Both techniques consume more-or-less the same amount of bandwidth. Participants were then asked to rate the statement “I would use this streaming technique when bandwidth is limited” using a 5-point Likert Scale \[49\] for both scenarios. After participants rated both techniques, the observer explained which technique we study in this thesis. Participants were given a short explanation on the approach of this use case. The explanation was ended by mentioning that the technique depends on some parameters and that he/she will then see a series of scenarios where these parameters are varied. For each scenario, the participants had to rate the statement “I would use the streaming technique in this scenario when bandwidth is limited” using the Likert Scale. After all the scenarios were finished, participants were asked about their overall idea of the technique.

![Figure 7.18](image)

Figure 7.18: A chart displaying a comparison between an overall lower quality video and a high quality viewport video both usable under limited bandwidth conditions. The comparison is made using four values: median, mode, minimum and maximum.

In Figure 7.18, we can see a comparison between two techniques that are usable under limited bandwidth conditions. The comparison is made between an overall lower quality video and our proposed solution, which streams the viewport in high quality and the rest in a lower quality. The chart compares the two techniques using four values: median, mode, minimum and maximum. We can see that both techniques have an identical mode, minimum and maximum. However, the median for our solution is slightly lower compared to the median for the overall lower quality. Because the median value is higher than 3 (which

---

2The value that occurs the most in a set of data.
is the rating for neutral), it means that people generally would like to use both techniques. With one-way ANOVA, we found no significant effect of the technique on the rating ($F(1,26)=4.90$, $p=0.03$).

Participants that rated the overall lower quality approach higher than our solution had the following feedback: “I am used to an overall lower quality”, “I did not see a lot of difference between the high quality and the overall lower quality” and “I prefer an overall lower quality compared to a harsh transition”. Other participants rated our solution higher and explained their preference with the following feedback: “I am not disturbed by the switching of qualities”, “I prefer a video with a quality that is as high as possible, thus I rather like to spend a short time seeing low quality than seeing an overall lower quality all the time”.

We can see that both techniques are acceptable when bandwidth is limited. The median of our solution is slightly lower than the other technique. Participants that prefer an overall lower quality, seem to be used to the concept and thus do not mind the result. However, users that want a high quality video, do not mind to watch a short period of lower quality video footage, to enjoy the rest in a higher quality. From participant’s feedback and interpreting of the results, we learned that our dataset for the tiling approach proved to be lacking. We had three possible qualities: high, medium and low. Each tile had a resolution of $600 \times 200$. The high, medium and low quality tiles had a bitrate of respectively 375, 125 and 32. The high quality was too high, while the low quality was too low. We could lower the bitrate of the high quality and still get high visual fidelity. It was hard to distinguish video objects when watching the low quality tiles. For future studies, it may be worthwhile to determine a balance between high quality tiles and low quality tiles. Balancing both would lead to less harsh transitions between tile qualities, which was a remark by participants that did not prefer our solution.

In Figure 7.19 we see a chart that displays the median for different configuration options of our solution. Each configuration option is named by its three important parameters: switching time, quality outside of the viewport and panning distance. This chart gives an overall impression of all the configuration options. We can see that each configuration that uses a medium quality outside of the viewport has a median value of 4. For low quality outside of the viewport, the median values vary. Therefore, we are only going to look at configuration options with a low quality outside of the viewport.

In Figure 7.20 we can see the median values for each configuration with a fixed value for the quality outside of the viewport (i.e., a low quality). We can see that there is a small difference between the small and medium panning distance and between the medium and big panning distance. The small distance is rated slightly higher than the medium distance and the medium distance is rated slightly higher than the the big distance. Switching times of 2 and 4 seconds are generally rated higher than the 6 seconds switching time.

With two-way ANOVA, we found a significant effect of the quality outside the viewport on the rating ($F(17,234)=381.61$, $p=8.44e-52$). We also found significant effect of the panning distance on the rating ($F(5,36)=381.61$, $p=0.02$). There was a significant interaction between quality outside of the viewport and the panning distance, therefore, we discarded the main effect of quality and panning distance and analyzed the interaction in detail. We fixed the panning distance (i.e., low, medium and big). For respectively low, medium and big, we
Figure 7.19: A chart displaying the median of the participant’s ratings for different configuration options of the high resolution viewport technique. Each configuration is named by the three important parameters: switching time, quality outside of the viewport and panning distance.

had the following results: with one-way ANOVA, we found a significant effect of the quality on the rating ($F(1,26)=23.26$, $p=5.36\times10^{-5}$), we found a significant effect of the quality on the rating ($F(1,26)=80.26$, $p=1.99\times10^{-9}$) and we found a significant effect of the quality on the rating ($F(1,26)=92.63$, $p=4.68\times10^{-10}$). In other words, there is significant difference between the ratings of configurations with a medium quality compared to a low quality outside the viewport for all panning distances (i.e., medium quality is rated high and low quality is rated lower). Even though, time seems to influence the results, the is no significant effect for time on the rating. This means that differences in time does not cause the rating to vary a lot.

Participants indicated that they generally wanted to use this technique when medium quality is used outside of the viewport. For the low quality case, they gave us the following statements: “tiles are too obvious”, “the low quality is too low”, “I do not mind the low quality when it switches fast enough”, “low quality is acceptable when the panning distance is not too big”, “the difference between high and low quality is too large” and “I would smooth the edges of the tiles to make them less obvious”. Participants explained that their rating would be higher when they could focus on specific objects in the video, assuming these objects are located in the high quality part of the video. In the current test, participants focused on the parts that were displayed in a lower quality. For future studies, a comparison could be made for this technique using videos with certain “focus objects” and a video without objects to focus on.

We can conclude that a medium quality outside of the viewport is rated very high. When using a lower quality, smaller panning distance and fast switching time is important. When creating the dataset, we should aim for a smoother quality transition and perhaps make sure there is enough video footage that a
Figure 7.20: A chart displaying the median of the participant’s ratings for configurations with low quality outside the viewport and with varying switching time and panning distance.

user can focus on.

7.6 Conclusion

When streaming high resolution videos (e.g. ODV) over a network, it can be troublesome in situations where bandwidth is limited. When bandwidth is not limited a video streaming application can opt to send the high resolution video. However, when bandwidth is limited, an application can adapt to the current situation and use a solution that tries to maximize the quality of the video under the present network conditions.

In this thesis, we developed a technique that tries to maximize the quality of the “important” part of the video. Because we use this technique in the context of ODV, we define the viewport as the important part. We therefore try to maximize the quality of the viewport and lower the quality of the video outside of the viewport. To be able to adapt to viewport changes, we divide the video into tiles. By using tiles, we are able to identify which tiles are part of the viewport. These tiles are then send in a higher quality compared to tiles outside of the viewport. In practice this means that the user sees the viewport in high quality. When the user pans, a lower quality will be visible for a short time. The application will then as soon as possible update this lower quality to a high quality.

In the experiments, we concluded that our technique saves bandwidth compared to sending a high resolution video. The consumed bandwidth of a tiled video stream is directly related to the number of tiles in the viewport and quality outside of the viewport. The lower the number of tiles and the quality outside of the viewport, the lower the consumed bandwidth. We also noted that because
of parallel requests for each tile, we are able to have a higher throughput. Users slightly prefer an overall lower quality compared to our solution, because they are generally used to an overall lower quality. In the user test, we saw that users would use this technique depending on the configuration options. Users like this technique when the quality outside of the viewport results in a watchable video. When video is barely watchable, they still accept the technique if the low quality is upgraded fast enough. Users do not like to see low quality tiles covering the viewport for a longer period of time. Participants notified us that they would probably rate our technique higher when “focus objects” were present in the high quality part of the video.

We used this approach in the context of ODV to be able to stream the viewport in high quality. However, this technique can be adopted for any video streaming scenario that wants to prioritize the quality of a part of the video over the rest of the video.

When comparing our approach with the discussed related work it is important to note that our solution uses a static number of tiles similar to the method described by Wang et al. The technique of sending low quality outside of the region of interest and upgrading the low quality when needed is similar to the technique proposed by Devloo et al.

The lack of a decent DownloadSegmentRateEnforcer makes adapting to viewport changes harder. Therefore, an improved implementation of this module would positively influence the functionality of this use case. The quality adaptation logic developed for this use case is rather basic. The logic sends the viewport in high quality and the tiles outside in a low or medium quality. A logic that would take the available bandwidth into account and thus possibly increase or decrease quality outside of the viewport accordingly, would result in better results. The quality of tiles in directly dependent of their location in or out of the viewport. As future work a technique could be developed to introduce more levels of quality. For example, tiles just outside of the viewport could be streamed in medium quality, to have a smoother transition between high and low quality tiles. As future work, we could also implement a feature to be able to identify important objects in the video. Even if these objects are out of the viewport, we could stream them in a decent quality. When these objects become visible in the viewport, the user would probably focus on these objects.

Under limited bandwidth conditions I would surely opt to use this technique when the switching time is fast enough. I would gladly compromise the quality of the video for a short time to than be able to watch it in a high quality.
Chapter 8

Multi-layer

8.1 Introduction

In this use case we try to find an alternative solution for the problem explained in Chapter 7. Sending high quality videos over the network introduces bandwidth problems. This use case has a lot of similarities to the use case of Chapter 7, in that we try to reduce consumed bandwidth by sending only certain parts of the video in high quality.

In contrast to the tiling approach, we want to provide only certain objects inside the video in high quality. These objects should be the probable points of focus when a user watches that video. As such, they are typically part of the foreground and contain a lot of visual detail. The rest of the video data is then marked as background and will be sent in a lower quality. The idea behind this approach originates from the computer vision field, where a lot of research is done in either real-time or off-line segmentation of foreground objects [75]. According to [75], foreground objects are objects that appear in front of a dynamic texture with distinctive statistics in space-time, for example ships on the sea and people riding an escalator.

In a multi-layer approach, we send different depth layers of video data over the network. Each of these layers is streamed in different video qualities. The front and back layers contain respectively the highest quality and the lowest quality video data. It is possible to introduce additional layers in between the front and back layer to extend the number of quality layers. In the case of two layers, foreground objects are played inside the front layer and sent in high quality. The rest of the data is part of the back layer and sent in a low quality. The to-be-applied quality difference between layers depends on the application scenario at hand.

In our case the multi-layering is an approach to reduce consumed bandwidth by sending the back layer in a low quality. The objects that users focus on are contained in the front layer. Other important objects without a lot of visual detail could be part of an intermediate layer. We make a preprocessing decision about which objects seem important within a video. These objects are segmented out of the original video using off-the-shelf 2D object tracking software. The rest of the video data is stored as a background video.

Sending these layers over the network should cause lower consumed band-
width. However, when they arrive at the client, the video should be reconstructed from its individual layers. Because each layer contains a certain depth level of the video, they should in essence be adequately placed on top of each other to recreate the video.

8.2 Related work

Goor et al. have proposed an adaptive MPEG-4 streaming system based on object prioritisation \[30\]. Their solution exploits the Video Object (VO) coding of MPEG-4. By applying priorities to objects, video adaptation can be done based on the content. The idea is that the priority of the object reflects the quality in which it must be perceived. High priority objects will be streamed in a higher quality compared to low priority objects/background. The paper describes two main features: segmentation, prioritisation and encoding of VOs on the one hand, and a client-server system for streaming this kind of data on the other hand.

The authors use different kinds of technologies to achieve their result. By using MPEG-4’s VO coding, coded representations of media object can be made. Media objects can be still images, video objects or audio objects. Multiple media objects may be represented by a single VO. For example, an actor as a video object and his voice as an audio object can be represented by one VO because they describe one object. Therefore, more than one Elementary Stream (ES) may be required per object. To be able to associate each stream with an object, the Object Descriptor (OD) is used. A VO is represented by a sequence of Video Object Planes (VOPs). A VOP is a video frame of a specific object of interest to be coded \[28\]. VOPs contain YUV texture matrices and shape matrices. To group multiple objects together, the Binary Format for Scenes (BIFS) is used by defining the position of these objects in space and time. MPEG-J defines a set of Java APIs to access and control the MPEG-4 terminal that is playing the video session \[?\]. By using these APIs responses can be send to the MPEG-4 terminal to for instance alter the description of the scene. MPEG-J applications are referred to as MPEGlets. MPEG-J is used for two operations: to alter the scene description to change or possibly omit objects from the scene and to monitor the capacities and resources for streaming videos to a terminal. MPEG-7 is used to describe the content by the means of meta data. The solution is scalable because each object can be modified separately to adaptively stream the content. A dataset for this solution can be made by segmenting of the video into objects. Once the VOs have been separated and defined, information about these objects is stored. Additional to this information, the priority of each object is also stored in the meta data.

Each VO is described by multiple ESs. To actually transmit these ESs they will be multiplexed at server-side. At client-side, the multiplexed stream is demultiplexed. Each ES is then decoded to get the object data and the meta data. The compositor uses the object data together with the BIFS information to arrange the video content. When the client wishes to play a video, an MPEGlet is transmitted to the client that provides this media clip. This MPEGlet is also responsible for monitoring and sending feedback about the capacities and

\[\text{In the YUV color-encoding scheme, the luminance and chrominance channels are separated.}\]
resources of the client video terminal. By reacting to changes in capacities or resources, the application is able to adaptively stream a quality that can be handled. If resources are scarce, the quality of the VOs will be lowered. If the degradation of the objects is too large, objects are omitted from the scene.

Hsiao et al. have described a object-based video streaming technique with application to intelligent transportation systems [32]. The system uses a background-registration method to separate vehicles from the background with an adaptive threshold. The proposed system consists of six key modules: the object segmentation, MPEG-4 encoder, sender, receiver, MPEG-4 decoder, Composer and Traffic Monitor. The basic flow of the video starts when it is captured by the surveillance camera. The video is than segmented with a background-registration method. By using an adaptive threshold, objects are separated from the background. The background-registration technique updates the background buffer to create a Binary Object Mask (BOM). This mask describes the location of the background. The next step is to use the MPEG-4 encoder to encode VOs in a higher quality compared to the background. The data is then streamed over a network using RTP. When the stream is received, it is decoded. After decoding, it is composited by using the BOM to identify which objects are foreground and which data is background. This video data is then used to do road traffic analysis to, for example, detect a traffic jam.

8.3 Approach

As with any multi-layer approach, to be able to stream a video in multiple layers, it first has to be segmented. To segment our videos we use the rotoscoping tool in Adobe After Effects [2]. To separate certain objects from the background, we make sure these objects are selected for the rotoscoping operation. The rotoscoping tool creates a new sequence of frames with only the selection visible. We will use this sequence of frames to create a foreground video. To create the background video, we take the inverse of the foreground video. The parts of either fore- or background that have no video data will be displayed in one static color. Currently we use bright green or black. In the remainder of this thesis we will refer to this as the chroma key. At this point we can decide which qualities we want to provide for both layer. Generally, we provide the foreground video in a higher quality than the background video. In Figure 8.1 we can see an example of the videos that are created by the rotoscoping tool. In this example, a bright green color is used as a chroma key to describe the overlapping areas. When displaying both layers, the bright green color should be eliminated by the application. However, we could also choose to still send the original video (at a lower quality) as the background video without segmenting out the foreground. The size of the background video would be bigger in this case but would allow for more flexibility in terms of reconstruction. When all encoded versions of both videos are available, we create two MPDs: one for the foreground and one for the background. These MPDs can be used to integrate the multi-layering in our implementation framework. We describe these MPDs in the media experience file in a similar way as in the use case in Chapter 7.

In Listing 8.1 we can see an example of a media experience description for
a multi-layer approach. The MPDs that describe foreground and background video are included in separate nodes. Both nodes are part of a group. The `activeOutWidget` attribute is used to describe the layer structure. The highest value for a stream in a group is the foreground. In this example, node with id `ind02` is the foreground. The idea is that streams should be displayed on top of each other according to the value of the `activeOutWidget`.

```
1 { 
  "nodes": [{
    "id": "grp01",
    "isRoot":true,
    "isGroup":true,
    "caption": "RootNode",
    ...
  },{ { 
    "id": "ind01",
    "caption": "Ind Stream 1 − Background",
    "mpdURL":"http://.../multi-layer/background/template.mpd",
    "activeOutWidget" : 0,
    ...
  },{ 
    "id": "ind02",
    "caption": "Ind Stream 2 − Foreground",
    "mpdURL":"http://.../multi-layer/foreground/template.mpd",
    "activeOutWidget" : 1,
    ...
  }],
  "edges": [ 
```
Because the back- and foreground streams are located in a group, they both are streamed to the frontend at the same time. The meta data for these streams tells us how they should be positioned. In the web-based solution, we use one HTML5 canvas on which each layer is drawn. To display the multi-layer video, the order of drawing the layers is important. We chose for a HTML5 canvas because it allows multiple draw calls to “recomposite the segmented video”.

When we draw the streams on top of each other in the form we receive them, we would only see the last layer drawn on the canvas. We already mentioned that a part of each layer has to be eliminated, for example the green color values in Figure 8.1. After eliminating the unwanted pixel values, drawing all layers on top of each other gives us the wanted result.

We use WebGL to eliminate the unwanted pixel values. Using WebGL gives us the ability to process each pixel on the GPU. For each pixel we determine if it should be eliminated or not. To eliminate the value of the pixel, we make the pixel transparent. We created two fragment shaders that eliminate the black or green chroma key. In Listing 8.2 we can see the fragment shader that is used to eliminated the bright green color from an image. In an ideal situation, we would have to filter out the pixels with RGB value (0,1,0). However, in practice we noticed that if we only eliminate these pixels, a lot of green color is still available. This can be explained by the lossy compression used to encode videos in this thesis. We determined a threshold that seems to eliminate most green colors. We also noticed that detecting non-meaningful pixels that form an 1 pixel edge around the foreground object is hard. We experimented with a more aggressive threshold that would introduce a 50% transparency for these pixels (line 12 in Listing 8.2). The problem with this approach is that it increases the probability that meaningful pixels are unjustly selected for elimination as well (i.e., the ratio of false positives increases). Pixels that are not detected, are wanted pixels and thus we can use their values as opaque output value (line 15 in Listing 8.2).

```glsl
precision mediump float;
uniform sampler2D u_image;
varying vec2 v_texCoord;

void main() {
vec4 textureColor = texture2D(u_image, v_texCoord);
if (textureColor.g >= 0.8 && textureColor.r <= 0.1 &&
textureColor.b <= 0.1) {
g1_FragColor = vec4(0.0, 0.0, 0.0, 0.0);
}
else if (textureColor.g >= 0.4 && textureColor.r <= 0.3 &&
textureColor.b <= 0.3) {
g1_FragColor = vec4(0.0, 0.0, 0.0, 0.5);
}
else {
g1_FragColor = vec4(textureColor.r, textureColor.g, textureColor.b, 1);
}
```

**Listing 8.1:** Example of a media experience description for a multi-layer use case
Listing 8.2: Fragment shader used when eliminating the green chroma key by making these pixels transparent

Currently the implementation is very basic and does not produce a perfect result. We either eliminate too many meaningful pixels or too few segmented out pixels. As an improvement we could eliminate less pixels and conceal the errors. The errors are caused by the chroma key border remaining around the foreground object. We could possibly look at edge smoothing to conceal these errors. In the current solution we eliminate too much pixels and make sure that unjustly removed data is also located in the background. As such, data should always be available for a pixel, albeit perhaps at a lower quality.

In the left example of Figure 8.2 we can see a frame containing visual artifacts that our implementation introduces using a black chroma key. Even though, the previous explanation was all based on a green chroma key, all these concepts are similar when eliminating the black color. To reduce these artifacts we increase the filter threshold. The right example in Figure 8.2 shows the output when we use this increased filter threshold.

![Artifacts](image1.png) ![Less artifacts](image2.png)

(a) Artifacts in the current implementation when only eliminating the pure black ((0,0,0) RGB) chroma key. Some false negative pixels remain and there is a small however, not very noticeable edge outline.

(b) Less artifacts are visible when we increase the filter threshold. However, with this filter threshold we also filter out correct data.

Figure 8.2: A frame of a multi-layered video in which the data for the actor is located in the foreground layer.

Because the output video stream is created out of multiple streams, all streams should play in a synchronized way. Similar to the use case in Chapter 7, we initiate playback of viewers in a viewerGroup at the same time. As a proof-of-concept implementation, this gives us more-or-less synchronous layer playback.

In the current implementation, we only allow for two layers: a foreground and a background. This allows us to send the foreground in a higher quality compared to the background. However, as future work we could include more layers, to improve the granularity of our solution by introducing intermediate quality levels.
8.4 Application

To view a multi-layered video in the frontend, we have to choose a media experience that describes such a video. We start to receive two streams and display them on top of each other. Each stream is stored in a viewer and is part of a viewerGroup. When each stream has buffered enough, the group will initiate playback of all its constituting streams at the same time. In contrast to a normal video player, the user sees a video with the foreground objects in high quality and the background in a possibly lower quality. In an ideal situation the user will not notice any difference in quality between foreground and background. Because a lot of bandwidth can be saved by using a multi-layered approach, users could find it acceptable even when difference between layers in terms of quality is noticeable. However, in our implementation there are still some unwanted visual artifacts.

In Figure 8.3 we can see a frame of a multi-layered video. This video recomposed out of two layers. The background is sent in a lower quality compared to the foreground. The remaining foreground data size depends on the spatial size of the foreground object in the current frame. We marked the red car as the most important object in the video. Therefore the car is segmented in the foreground layer. The rest of the video footage is stored in the background layer. The amount of high quality video footage depends directly on the size of the car in current frame. When the car is clearly visible we see more high quality data.

![Figure 8.3: A multi-layered video with the red car as priority object. The car is sent in a higher quality than the background. The green chroma key pixels are filtered out of the foreground layer.](image)

Our application allows us to display a multi-layered video. Even though some visual artifacts are visible, our application shows the basic concept of recomposing layers to create a multi-layered video. In Section 8.5 we will take a look at how much bandwidth we can conserve with this solution and make a comparison between using bright green and black as filter color.
8.5 Results

The multi-layering solution is created to conserve bandwidth when sending high resolution videos. We compromise the quality of the background to be able to send the foreground in a higher quality. In this section we will quantify how this solution behaves compared to sending the non-segmented video. We use two different colors to filter out non-meaningful pixels: bright green and black. We will have a look at how these two colors compare to each other in terms of visual accuracy of the multi-layer rendering result.

The experiment described in this section were are executed on the following setup: MacBook Pro 13-inch with Retina-display, OSX 10.9.4, 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3 RAM and Intel Iris graphics card. We used the Google Chrome browser version 39.0.2171.71 (64-bit) as an environment to conduct these tests in.

8.5.1 Multi-layer bandwidth consumption compared to high resolution videos

We would like to compare our solution with sending a conventional (i.e., non-segmented) high resolution video. To test this, we used a test case with two layers. By comparing the consumed bandwidth of a multi-layered stream and the corresponding non-segmented stream, we should see how much bandwidth we conserve with this solution. The bandwidth of a multi-layered stream is dependent on the amount of foreground data and the quality of the background layer. Therefore, we conducted this test for different kinds of foreground layers and different qualities of background layers.

As a dataset we used two separate segmentations of the same video. In the first video the actor is located in the foreground layer. The actor is in every video frame but only covers a small portion of the screen. In the second video the foreground layer contains the red car. This car is clearly visible for a short period and covers a small amount of pixels for the rest of the video. In both streams we provide two qualities for the background layer.

We expect the multi-layering to result in lower consumed bandwidth. The amount of consumed bandwidth is related to the amount of data in the foreground layer and the quality of the background layer. Therefore, we expect that when a lot of data is present in the foreground layer, more bandwidth will be consumed compared to having a small amount of data in the foreground layer. We also expect: the lower the quality of the background layer, the lower the consumed bandwidth.

![Figure 8.4: An example of a consumed bandwidth chart for a high resolution video stream. Streaming this video in limited bandwidth conditions can be troublesome. This non-segmented video is used to create two separate segmentations for this test.](image)
Figure 8.5: Two examples of multi-layered video streams. The foreground layer contains the actor, who is always visible. The background and foreground layers are displayed in a stacked graph and are respectively represented by ind01 and ind02. In the two examples we varied the quality of the background. The left example has a lower background quality.

Figure 8.6: Two examples of multi-layered video streams. The foreground layer contains the red car, which is visible for a short period of the video and covers a small portion of the video the rest of the time. The background and foreground layers are displayed in a stacked graph and are respectively represented by ind01 and ind02. In the two examples we varied the quality of the background. The left example has a lower background quality.

In addition to the stacked graphs used to display the results of the tests, we calculated the amount of bandwidth is consumed in each situation. To calculate this, we summed up the file sizes of the segments requested in each scenario. In Figure 8.4 we can see an example of the consumed bandwidth of a high resolution video stream. The amount of consumed bandwidth is 35.6 MB. We use this chart as a baseline to compare our multi-layer solution with. In Figure 8.5 we can see the consumed bandwidth of streaming of one of the corresponding multi-layered video. The consumed bandwidth for the left and right stream are respectively 7.6 MB and 9.1 MB. In this video we segmented the actor, who is always visible but only covers a small part of the screen. By lowering the background quality we see that even less bandwidth is consumed. In Figure 8.6 we can see two charts that represent the consumed bandwidth of streaming the other corresponding multi-layered video. The consumed bandwidth for the left and right stream are respectively 14.6 MB and 16.1 MB. The foreground layer contains the red car, which is clearly visible for a short period and only covers a small portion of pixels the rest of the video. The results of these charts are comparable to the charts in Figure 8.5. The amount of foreground data varied in both multi-layered results. When a lot of data is present in the foreground layer, more data had to be streamed. Compared to the high resolution stream, both multi-layered streams consumed less bandwidth. When the background layer quality was lowered, even less bandwidth is consumed.

We can conclude that less bandwidth is consumed when using a multi-layered
video. The magnitude of the bandwidth savings depends on the amount of foreground data and the quality of the background. When more data is located in the foreground layer, we consume more bandwidth. This also means that a larger portion of the resulting video is in a higher quality. We can see that lowering the quality of the background leads to consuming less bandwidth. A consideration has to be made how many objects should be in the foreground layer and what kind of quality the rest of the video should have.

8.5.2 Comparison between bright green and black filtering

To make a comparison between bright green and black filtering, we will look at the advantages and disadvantages of filtering with each color. We can then decide on the most suited color for each scenario.

This comparison will give us an idea what color we should use to filter on in our scenario.

First we discuss filtering with each color in practice. Bright green is rarely present in real-life video footage. Therefore, it can easily be filtered out a video without filtering out false positives. However, when only a very small portion of the green color remains visible in the reconstructed video, it will be clearly noticeable. Black on the other hand is often present in a real-life video. When we filter out black, we often also disregard pixels that contain useful data. When a small border of black pixels remains around a foreground object, it is less noticeable compared to a bright green border.

To make sure we filter all the non-meaningful pixels out, we use a thresholding technique. Because bright green is rarely found in a normal video, we can use a more aggressive threshold without running major risks that meaningful pixels will wrongly be filtered away. The downside to using green is that we always have to use a higher threshold to make sure all bright green colors are gone since they are very noticeable in the recomposed result. Using thresholding with black as a filter color can be more troublesome. Since black is often present in a video, we must resort to a more conservative threshold value to prevent relatively large numbers of meaningful pixels from being selected. The advantage of filtering out the black chroma key pixels is that even a conservative threshold results in a plausible result.

When we filter out a part of the correct data, we introduce areas without pixel data. A possible solution to this is sending the entire video in a lower quality as a background layer and not the inverse of the foreground segmentation. When we send the entire video, each pixel in the recomposed video will always have meaningful value. It may be possible that some areas of the reconstructed video end up with a lower quality than intended. This means we can use a higher threshold to filter out unwanted pixels while keeping data for the correct pixels. This can be helpful for both colors.

We talked about all the advantages and disadvantages of filtering with each color. However, it is also important to look at the file size of the videos with the foreground layer details. The file size varies between a black and bright green filter color. We used two different kind of segmentations. The first is a very short video where on average 60% of the actual video footage data is located in the foreground layer. In case of the green color, the file size is 1.8 MB. When using the black filter color, the file size is 1.3 MB. The second video is a longer one, where the foreground layer contains one object. This object is only clearly
visible for a short period and only occupies a small portion of pixels the rest of the time. In case of the green color, the file size is 10.2 MB. When using the black filter color, the file size is 9.9 MB. We can see that when the foreground layer contains more actual video data, the difference between black and green is bigger. We suspect that this is caused by the encoder being able to more efficiently encode the switching between blocks of black pixels and meaningful pixel compared to blocks of green pixels and meaningful pixels. When the foreground only contains a small amount of video data, the file sizes are not very far apart. However, using a black filter color always results in a lower file size.

We can see that each color has its advantages and disadvantages. It mostly depends on the kind of application at hand. When we want to filter out a color in a colorful video, we can get away with using black as a filter color. This means that the resulting file size will be lower. However, when we want to use black in a video with darker colors, we will maybe filter out too many false positives. Bright green is easily filtered out of a video but is very noticeable when only a few pixels are not correctly filtered out (e.g., when a border remains visible around the foreground object caused by lossy compression). When high thresholding is permitted we can use the bright green color because it is less likely that the color is used in the actual video footage.

8.6 Conclusion

Streaming high resolution videos in limited bandwidth conditions can be troublesome. These conditions can occur when for example using a mobile Internet connection to stream the videos. We therefore looked for a solution for the problem with the large amount of consumed bandwidth when streaming high resolution videos.

In our approach we select a part of the video to send in high quality, while sending the rest in a lower quality. We create multiple layers which correspond with different depth levels in a preprocessing step. We select objects that may be of interest to the user and offer these in high quality. Because these objects generally have the focus of the user, we send the rest of the data in a lower quality. In practice this means that part of the video will be in a high quality while the rest is in a lower quality. We send each layer separately over the network link and thus need to reconstruct the video at client-side. Filtering is used to select pixel values for the resulting video. In the current implementation it is possible to use two filter colors: black and bright green.

We found that our solution lowers the amount of consumed bandwidth as expected. The amount of consumed bandwidth is directly related to the amount of foreground data and the quality of the background layer. The decision to do pixel filtering with either black or green depends on the used scenario. Each color has its own advantages and disadvantages.

The suggested solution can be a used for any video streaming scenario where part of a video is more important than the rest. When working with limited bandwidth conditions it is always helpful to reduce the consumed bandwidth.

Our solution closely resembles the Video Object coding of MPEG-4. In this kind of coding, multiple object layers are created and later reconstructed. When using VO coding the video can be reconstructed using objects of a different video
quality, which is similar to our approach.

In the current implementation we still have some artifacts in the reconstruction phase. To have an ideal solution, these artifacts should not be present in the resulting video. When using a green filter color, a small border is present around the foreground object. We can use a high threshold to filter this out of the video but then risk adjusting correct pixels in the process. Currently, we only implemented a solution to send the full background video in a lower quality. In this case we always have data present for each pixel. However, a more advanced filtering technique can result in less artifacts. Even when artifacts are present we could implement an error-concealment algorithm, for example interpolation or edge smoothing, to reduce artifact impact on the visual fidelity of the reconstructed result.

The quality of the resulting video is heavily dependent on the segmentation of the original video and reconstruction of the layers. Our solution shows the concept of multi-layering even with a basic filtering technique. In my opinion multi-layering is a good solution for bandwidth problems in certain situations. However, reconstruction has to be close to ideal to use this kind of technology in video streaming applications. An other downside is in order to create a decent segmented video, we primarily have to resort to an offline segmentation of the video.
Chapter 9

Multi-Screen

9.1 Introduction

Up to this point, we have always talked about watching an ODV through a spatially limited viewport. In this use case, we will explore the idea of extending the viewport over multiple screens. When using only a single screen, the information within the viewport cannot extend a certain limit if we do not wish to keep zooming out. Without zooming, we can extend the viewport if we add multiple screens to draw the viewport on.

The location of the screens is important in such a setup because this will determine in what way we extend the viewport. If screens are positioned in an intelligent way, we can imitate an immersive experience comparable to those delivered by a CAVE environment. The CAVE is a cube-shaped virtual-reality room, whose walls, floor and sometimes ceiling are entirely made of computer-projected screens [19], [17]. Projection-based VR systems, such as CAVEs, provide surround viewing. Sometimes fully surround viewing is not supported; in those cases, at least surround viewing in 90 degree in each direction has to be present to not see the edges of the display. These systems track the user to provide the correct perspective rendering [19]. In Figure 9.1 we can see an example of a CAVE setup. We clearly see it gives a different experience compared to viewing information on a single 2D computer screen.

To elaborate what we mean with intelligent positioning the screens within the multiple screen setup, we use an example. In Figure 9.2 we see an example of how 3 screens can be positioned. We consider this an intelligent positioning because, it will extend the existing viewport from the left screen to the right over two other screens. This approach will let the user view additional information to the right of the original viewport. Generally, screens that are located side-by-side, are considered a intelligent positioning.

When screens are not located next to each other, a user will receive a different kind of experience compared to an immersive CAVE-like experience. This approach can for example be used in cases where we want to always see two fixed viewing angles within the ODV. In this thesis, we will consider the multi-screen approach in the case of intelligent positioning of the screens, with the goal to improve the immersiveness of the setup.

Similar to the tracking of the user in projection-based VR systems, we have
Figure 9.1: An example of a CAVE setup. This CAVE setup is located in the National Center for Supercomputing Applications at University of Illinois at Urbana-Champaign [19].

Figure 9.2: A example of how three screens can be positioned within the multi-screen setup.

to provide correct perspective rendering in multi-screen settings. In this use case, the correct viewport content has to be rendered on each screen. In the case of an ODV, a part of the viewport has to be displayed on each screen.

In Section 4.5 we explained which interaction possibilities the ODV Viewer provides. Interacting with the ODV over multiple screens should provide the same functionality. Interacting should not only change information in one screen
but should change the information in the other screens accordingly. For example, when a user pans left, he will see video footage that was located to the left of the original viewport. Because the viewport is spread out over multiple screens, this interaction should result in a shift of the viewport over these screens.

9.2 Related work

Combining multiple heterogeneous displays can serve various purposes. It mostly comes down to providing more information to the user, which would not be possible on a single screen. In this thesis, we make the assumption that these displays do not have to be connected to the same computer. This can make the interaction with multiple screens more challenging. A mouse and keyboard are generally connected to one computer. If we would like to use this method of interacting, we would hence need to connect a mouse and keyboard to every display’s computer. This rapidly introduces a “mouse (or keyboard) jungle” [66]. In some cases, synchronization of the information displayed on the multiple screens is required. A good example is a scenario where a video must be displayed across multiple displays. When separate computers are used in a setup, we have to find a way to synchronize the content. In this section, we will take a look at what approaches are used to interact with multiple screens, what kind of information is shared among them, and how synchronization is done.

Booth et al. have described a solution for interacting with multiple heterogeneous computers that are viewed simultaneously, called “Mighty Mouse” [11]. Their solution is created on top of VNC’s network protocol [59]. The x2vnc implementation of the VNC protocol [34] already provided a way to allow controller/controllee pairing and to control multiple controllees in sequences. Booth et al. based their work on this idea. They used a VNC server on the controllee to handle all the interaction with the host and a VNC client on the controller. However, they provide a multi-platform solution that improved the interface of the controller and provide the ability to control and adapt to multiple controllees. In their implementation, every controller can be connected with an idle controllee. Navigating between these screen can be done through a user interface to directly go to the selected screen or by moving the mouse to the border of one screen, which will make the mouse appear on the screen that is spatially located on that side of the border. They allow for cross-platform cut and paste of text in addition to interacting with multiple cross-platform displays. This technique can for example be used in meetings, where multiple people show information on a projector. Using this way of interacting, multiple people can interact with the screens that are projected.

Rekimoto has proposed a pen-based direct manipulation technique, called Pick-and-Drop, that can be used for data transfer between different computers as well as within the same computer [60]. This technique lets users pick up and drop objects from one display onto another as if they were manipulating physical objects. The author states that transferring data between computers over the network is rather cumbersome. A simple cut-and-paste is handled very different for distributed hosts than on a single computer. The interaction tool within this paper is a stylus, comparable to a WACOM stylus. The idea is that the interaction of this stylus tries to remove the boundaries between multiple computers. To be able to pick and drop objects, each pen is assigned a unique
ID. Interaction done with the pen will be monitored by a server called the “pen manager”. When a pen picks up an object, the manager will bind the object’s id to the pen id. The object can then be dropped onto another computer. The interaction is done by tapping the pen on the screen. Because the assumption is made that all computers are connected to the same network, the pen manager can then “drop” the item onto the other computer.

Johanson et al. have described a framework that extends the information browsing metaphor of the Web across multiple displays, called Multibrowsing [39]. The users are able to move web pages or linked information across multiple displays. In their framework a display can assume three roles: regular client, enhanced client and target. A regular client can direct a webpage to other displays in the environment by sending multibrowse fat-links, which contain information about the target, the display option and the webpage. An enhanced client has the same functionality as the regular client but is able to discover targets in the environment. An enhanced client can also pull a webpage to itself from any enabled display in the environment. A target can have webpages directed to itself. These webpages can than be displayed on the target or “pulled off” by enhanced clients. A regular client can use any webbrowser; an enhanced client has to install a custom browser plug-in called MB2Go and targets run a special service called butler service [63]. The butler service allows for the dynamic detection of available target displays. Since each target has the butler service installed, a list of discovered targets can be made. MB2Go allows enhanced clients to select a target from the discovered targets list. When a selection is made by the user, an event is sent that will make the butler service display the web page.

Huang et al. have presented a frame synchronization mechanism for interactive surrounding display environments that are constructed by multiple rendering hosts [33]. Their goal was to assemble a system with high performance but inexpensive to create, with immersive features comparable to those of a CAVE. The authors used a PC cluster to render images for each screen separately. A tearing effect appears if these images are not displayed at the same rate, this means they are out of synchronization. They provide a solution for this synchronization problem. Their approach consists out of two phases: a parallel phase and a phase interface. In the parallel phase, slaves render in parallel, while the master prepares the next frame to be rendered. Each node waits for the slowest one before entering the next phase, they call this the Convoy effect. When entering the phase interface, all slaves have rendered their part. Thus it is time for the slaves to receive data for the next frame. After the phase interface, the system returns to the parallel phase by telling the slaves to rasterize the previous frame and render the new one with the newly received data. This is called the Marching effect. The authors created a controller to synchronize the Convoy and Marching effect. The controller informs the master when the Convoy effect is done. The master can then trigger the Marching effect to enter the next parallel phase. The previous frame will then be displayed on each slave and each slave will render the new frame. The controller can be implemented with a hardware or software approach. A hardware approach, with a parallel port, is more accurate than the software approach on a local Ethernet network. However, a software approach still provides acceptable performance results.

Nam et al. have described two algorithms to synchronize displays in ultra-high-resolution display walls [54]. Their solution supports the synchroniza-
tion for multiple applications with varying frame rates. A lot of solutions are available for synchronizing one application over a cluster of PCs, for example, Chromium [35] and Equalizer [22]. These solutions are able to synchronize displays by using a "simple" network barrier, which waits until it received a message from all nodes before letting these nodes send data over the network. For instance, the Message Passing Interface (MPI) [48] supports a barrier over all communication nodes. The authors provide two algorithms to provide synchronization between displays for multiple applications at the same time. This synchronization can be split into three parts: data synchronization, swap buffer synchronization and synchronization of the vertical refresh cycles of the various displays. Data synchronization means that each display must display a part of the same frame at all times. Swap buffer synchronization is achieved by letting each node swap the content of the graphics buffer synchronously to display applications consistent across nodes. Synchronization of the vertical refresh cycles of the various displays occurs when the physical refreshes of each monitor happens synchronously (i.e., at an identical rate). The first two synchronization parts are handled by the algorithms while the vertical refresh cycles synchronization can be achieved through specialized hardware. They proposed two algorithms: a two-phase and a one-phase algorithm. Both algorithms use the same synchronization manager to be able to accomplish data synchronization. To maintain data synchronization, a single global synchronization master is used. It will determine a update rate that is greater than the highest update rate of one of the applications. This makes sure each application can be updated in time. In the two-phase algorithm, swap buffer synchronization is done by a network barrier after the data synchronization. The one-phase algorithm uses the Network Time Protocol (NTP) to synchronize clocks on each node. The one-phase algorithm has higher synchronization accuracy but is more CPU-intensive and the cluster has to support NTP.

9.3 Approach

To be able to stream videos to multiple screens, we made a specific design choice. We make the assumption that each screen displays a part of the viewport of the same ODV. As such, it is sufficient to have the ODV data on one client and forward it to the others. The viewport in these screens then has to be adjusted to display the correct part of the video. In this approach, we reuse the middleware written for the gesture-tracking server, which we explained in Section 5.3. This middleware handles all the connections between screens and makes sure that data as well as meta data is broadcasted to all of them.

To be able to display videos on multiple screens, we run multiple clients of the same application. Every client has two connections to the middleware. Only one client makes a connection with the backend similar to the use case in Chapter 6; we call this client the master. After some initial setup, the master will receive media data from the backend. Upon receiving this data, he will store it in a viewer and forward it to the middleware. The middleware makes sure to forward it to all the other clients, we call these the slaves. In Figure 9.3 we can see a diagram that shows all the connections between components in the implementation framework. We can see that the master has one connection to the backend and two to the middleware. The slaves only have the two connections.
to the middleware. Because we reuse the middleware from the gesture-tracking server, we are able to interact with our solution using gestures.

The master controls the data that is being played and what part of the ODV frame each slave displays. Connections are made to the middleware for two reasons: to avoid stressing the master too much and to have a single point of synchronization in the setup. However, the master will always have the final decision on actions within this setup. In this case it makes sense that the master opens a connection to the backend to receive all the data. We make the assumption that the master, slaves and the middleware are connected to the same network, thus we can send data over the local network. In a real-life scenario, it is likely that the data sent from and to the backend is transported over an Internet connection. Because an Internet connection is probably slower than a local network connection, we try to avoid opening too much connections over the Internet. We thus chose to open only one connection between frontend and backend. An alternative would be to open multiple connections, one between every involved client (either master or slave) and the backend, but this would cause a lot of traffic to be sent over the Internet, thus data would arrive slower at the frontends. In our case it is sufficient to open one connection to the backend because each client needs a different part of the same ODV frame. When the ODV is available in one client, the master, we can make it available to every other client. To be able to play a spatial portion of this ODV frame, the slaves only need guidance from the middleware with regard to viewport framing, thus they almost do no extra work.

We have chosen to connect every client to the middleware, rather than connect every slave to the master. The middleware has two connections to every client: one channel to send control messages and one to send data messages. Connecting all the slaves to the master would introduce more logic in the master, thus increasing the load on the master. In the current implementation, the added work for the master is forwarding the media data over a socket connected
to the middleware. Because new connections are handled by the middleware, it is possible to let new clients join the setup dynamically. Each slave will receive the same kind of information. Each slave receives the same ODV media data, but control messages can vary. Some information may be specific to the spatial location of the slave’s screen. Every client will handle control messages the same way. Only the slaves will receive data messages from the middleware. The middleware only accepts specific control messages from each kind of client. For example, a master can send the play event while the slave cannot.

Control messages between frontends and gesture-tracking server are different than those between backend and frontend. Control messages in this case are more related to the interaction with the clients. For example, there are messages to indicate panning and zooming. By using the control messages in an efficient way, we can create a setup for streaming an ODV over multiple screens. Some control messages are required for initial setup of the multi-screen approach. Data messages will in contrast to control messages have the same structure as those exchanged between backend and frontend. This is the case because messages are immediately forwarded to the middleware upon reception by the master.

Each screen in the multi-screen setup displays a different part of the ODV frame. We use an initial setup phase to determine which role the clients have: master or slave. Every slave provides the spatial location of its screen in relation to the master to be able to adjust the viewport. The master describes its relative location to the gesture-tracking camera. If no camera is active, the master’s location can be set as the initial location of (0,0). After an initial setup of the viewport, interaction on one screen will cause the same reaction on the others. For example, when a user pans, all viewports will pan in the ODV. This means that the middleware makes sure that the interaction is synchronized between clients.

Because each client displays a part of the global viewport, we want each client to play the video synchronously. Because each client can be run on a different computer, we need a mechanism to synchronize playback. Because each client is connected to the middleware, we can use this as a single point of synchronization.

The multi-screen solution is currently still under development by EDM. Therefore, not all functionality is present in the current implementation. In the current implementation we are able to select a role for each client. The master will make a connection to the backend and two to the middleware, while the slaves only connect to the middleware. Each client can configure its spatial location within the setup and send this information to the middleware. The middleware determines the viewport configuration for each client (i.e., the initiate location and zoom factor). The master is able to receive data from the backend and forwards this data to the slaves through the middleware. The backend can forward media data as well as meta data (e.g., minimal buffer time). When a client has buffered enough data, it sends a canPlay message to the middleware. Once the middleware has received the canPlay message from each client, the master can initiate playback for the setup. We use this kind of structure to have a more-or-less synchronized initial playback time. Once the setup starts playing the video, each client will play more-or-less synchronously.

Currently, interaction while playing the ODV is not yet implemented. The possible methods of interaction will be: gestures, and mouse and keyboard.
Gestures can be integrated with this setup because we reused the middleware from the gesture-tracking server. Mouse and keyboard actions would have to be intercepted and forwarded to the middleware. When the middleware receives an action (e.g., a panning action), it can send the appropriate message to each client.

This project is made in collaboration with EDM. Therefore, we did not implement all features of this solution ourselves. What we contributed to the project is the following: a framework to play ODV DASH content in a web browser environment, a connection library, which is used to connect each client to the middleware, and the idea for the synchronization technique, which is similar to the technique used in the use cases in chapters 7 8. The rest of the functionality is developed by other team members at EDM.

9.4 Application

In collaboration with EDM a web-based application is created which can be used to include multiple clients for a multi-screen approach. In this application we can define the role of the different clients. Every client has to provide the spatial location of its screen. The master defines its relative position to the gesture-tracking camera, while the slaves define their position relative to the master. These positions are used to calculate an initial location of the viewport within every client. To be able to receive data, the master has to connect to the backend. The master receives some meta data about the stream and media data for this stream. The master forwards all the necessary data for the multi-screen approach to the middleware. After initial buffering is done, the master is able to initiate playback for the multi-screen setup.

Figure 9.4 shows the setup options for a client in the multi-screen solution. We can see checkboxes to assign the role of the client: master, slave and stand alone. The stand alone role represents a client that is similar in functionality to a client in the use case in Chapter 6. Below these checkboxes we can see parameters that specify the spatial location of the client’s screen. This information is required by the middleware to calculate the viewport for each client.

![Figure 9.4: Setup options in a web-based client for the multi-screen setup. The client’s role and spatial location can be specified with these options.](image)
To give an idea of how we can initialize the setup and actual play an ODV, we will now describe a normal playback scenario. First, we make sure the middleware is running and accepting connections. Afterwards, we first setup the spatial configuration for each client. Once the setup is complete, we connect the master and afterwards the slaves to the middleware. At this point, all the clients are ready for the multi-screen approach. To be able to play an ODV, the master has to connect to the backend to begin receiving data. In the backend, the ODV is described by a media experience file. The master receives meta data (e.g., minimal buffer time) and media data for the ODV. Upon receiving this data, the master forwards it to the middleware. The middleware in turn forwards it to the slaves. When a client has buffered enough it sends a `canPlay` message to the middleware. Once all clients have sent a `canPlay` message, the master can dictate to begin playing.

In Figure 9.5 we can see an example of a physical setup on which the multi-screen approach can be run. All these screens are connected to a separate computer. The left, middle and right screens respectively have the following roles: master, slave and middleware. The master and slave are configured to display a continuous viewport of the ODV. In this setup, with only one master and one slave, the viewport can already hold double the amount of information compared to a single screen solution.

![Multi-screen setup](image)

**Figure 9.5**: An example of a multi-screen setup used at EDM. The left, middle and right screens respectively have the following roles: master, slave and middleware.

## 9.5 Results

We use the multi-screen approach to display more video information to the user at once. In our setup, the master has to perform some additional work compared to a normal client. To be able to quantify this, we will measure the work done by the master in a multi-screen approach and in a stand alone approach. In Section 6.5 we provided results for a path-based solution. This use case in essence does some additional work on top of the path-based use case. Therefore, the added stress on the master in a multi-screen approach will in this section be measured.

We conducted the tests on the following setup: one master client, one slave client and one middleware. The backend is ran on the same machine as the master. The DASH dataset is also located on the machine of the master. The clients, the backend as well as the webserver that hosts the media data are
connected by a local area network. Each client has the following specifications: Windows 8.1 Enterprise, Intel Core i5-4250U CPU 1.3GHz, Intel HD Graphics 5000, Intel Ethernet Connection I218-V 1Gbps and Google Chrome version 40.0.2214.38 beta-m (64-bit).

9.5.1 How much added stress does the master have in a multi-screen approach?

Compared to a single screen setup, the master has to do some added work. We would like to quantify how much added work is done. To measure this, we profile the master client with the Google Chrome profiler tool. We measure two client configurations: a multi-screen setup and a stand alone setup. The difference between these two is the work added by the multi-screen approach. We do not need to measure the performance of the slaves, because the added work caused by processing messages from the middleware is small. For each configuration, we measured the performance while segments from the backend are received. We chose this time frame because the master/stand alone client is stressed the most during this period. This time frame varied for both configurations. For each test we measured the amount of work done in key functions. The key functions in this test are:

- BFCommunication::onMessage, receive media data from the backend
- Base64Decoding, decode the base64 encoded media data message
- GTCommunication::sendMessage including sending data over the socket to the middleware, sending a message to the middleware using a websocket
- drawImage, draw a video frame to the canvas.

This test gives an idea how much added work is done to be able to extend the ODV viewport over multiple physical displays. Because the master only has to forward the media data once to the middleware, the CPU consumption of the master is not dependent on the amount of slaves.

We expect to see added work by introduced sending data to the middleware. We expect that this amount is comparable with the amount of work done by receiving data from the backend.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Master + Slave</th>
<th>Stand alone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run1</td>
<td>Run2</td>
</tr>
<tr>
<td>BFComm</td>
<td>801</td>
<td>833</td>
</tr>
<tr>
<td>Base64Decoding</td>
<td>767</td>
<td>723</td>
</tr>
<tr>
<td>GTComm</td>
<td>808</td>
<td>794</td>
</tr>
<tr>
<td>drawImage</td>
<td>330</td>
<td>241</td>
</tr>
<tr>
<td>Time frame</td>
<td>8511</td>
<td>7985</td>
</tr>
</tbody>
</table>

Table 9.1: Results of four tests of the multi-screen approach. In the first two tests we used a master-slave approach. In the last two we used a stand alone client, thus did not send any data to the middleware. All the results are displayed in milliseconds (ms).

In Table 9.1 we can see the results of four tests of the multi-screen approach. In run 1 and 2 we measured the performance of the application when using a
master-slave approach. In these tests the master forwarded the media data to the middleware. In run 3 and 4 we measured the performance of a stand alone client without sending data to the middleware. In run 1 and 2 we spent around 800 ms in the functions to send media data to the middleware. The time frame of run 1 and 2 is on average 2.2 seconds longer, thus in these cases it took longer to receive all media segments from the backend. We can see that we spent more time in the drawImage function in run 1 and 2. We can explain this because in both configurations, the clients spend some time buffering the video. The time frame for run 3 and 4 has almost ended before data is actually drawn on the screen. In run 1 and 2 more draw calls are done because it is a longer time frame. To clarify, we will assume that each client spends the first four second of the time frame buffering. This means that in run 3 and 4, we will only draw video frames on the screen for two seconds before the time frame ends. In run 1 and 2 we spend four seconds actually drawing frames on the screen.

We can conclude that the workload of the master has certainly been increased. In total it took 2 seconds longer to receive all the segments. However, we have to remember that the master only has to forward media data once, thus the solution is scalable in terms of concurrent slave count. We have to make the consideration whether the additional video footage shown in the viewport outweighs the extra work done by the master.

9.6 Conclusion

Our solution provides a way to increase the spatial extent of an ODV viewport using multiple screens. These screens allow the user to have a different experience compared to using a single screen. When the setup of the multiple screens is chosen wisely, the viewport of an ODV can be spatially distributed over these screens.

In this use case we reused already existing components to quickly draft a solution. In particular, we exploited the playback functionality of the use case in Chapter 6. We also reused the middleware created by EDM to do some processing. Another advantage is that this middleware allows us to use gestures to interact with the multi-screen setup. To be able to handle multiple clients, we connected each of them to the middleware. One client is designed to act as the master in this setup. This client receives data, displays it and forwards it to the middleware. The other clients are slaves. They receive data and display it. The master dictates the flow of the multi-screen approach. The middleware will broadcast messages, which the clients have to process and react to (e.g., panning).

The master client has to do some additional work. By quantifying this additional work we learned that we should consider if the additional video information outweighs this added work.

This approach can be reused to display additional data in all sorts of use cases. Because the playback scenario is mostly left untouched by the presented multi-screen approach, we could incorporate this with the tiling technique of the use case in chapter 7. Other kinds of displays like, for example, a display wall can be used as well. This would yield a different experience compared to those provided by the proposed multi-screen solution.

Synchronization is not optimal in our solution. In the future, we could
for example use an NTP approach similar to [54]. Currently only one way of interaction is incorporated, using gestures. We could apply a similar technique to [11] to navigate with the mouse and keyboard. Also, at the moment, screen positions are static. A possible extension could be to add support for dynamic screen locations. By adding this, movable tablets or smartphones could be used to display parts of the ODV on. Mobile devices could then be used as a kind of peephole display. Moving around the mobile device would change the viewport of the ODV.

In my opinion the current solution provides a good way to extend the spatial reach of the ODV viewport. Because this technique combines two new techniques for non-experienced users (e.g., ODV and multi-screen video viewing), it may take some time getting used to the concepts. Currently, most users are not familiar with the concept of ODV, therefore displaying it over multiple screen can be overwhelming. However, once users are familiar with the concept of ODV, I think they would appreciate viewing an extended viewport. This will enable them to see more video footage while keeping the same level of detail that they are used to. The approach is flexible because it is built upon already existing functionality. It could just as well be extended to a different use case.
Part III

Conclusion
Chapter 10

Conclusion

By using the MPEG DASH standard in a Web browser environment we were able to create four advanced video streaming use cases. By utilizing web browsers as execution environments, our proof-of-concept demos are intrinsically cross-platform.

In the first use case, we studied how MPEG DASH could be used to display a non-linear video. We utilized a technique called precaching to be able to fluently stitch the individual part of the non-linear video. We created a proof-of-concept demo that is able to provide a personalized non-linear video for each user. Because we were aiming for a usable solution, we provided results that our application is able to seamlessly switch between sequences in a non-linear video. A user study involving 14 participants revealed that, even if users can see a small transition between videos, they generally accept it as a stutter. We can conclude that because of the bandwidth usage and the decoding of incoming messages, that the precaching influences the performance of the application. However, it is necessary to provide seamless switching functionality. The choice of precaching algorithm to use was found important when looking at the performance of the application.

Working with high resolution videos in limited bandwidth conditions has proven to be troublesome. Therefore, we created two solutions that each limit the amount of high quality data sent over the network. Both solutions mark a part of the video to be sent in high quality, while sending the rest in a lower quality. The first solution uses tiles to be able to select a spatial portion of the video. By adjusting the quality of each tile, we are able to lower the consumed bandwidth of an application. The downside to this is that a part of the video is downgraded. In the context of an ODV, we used high quality tiles to represent video data that is located inside the current ODV viewport. Tiles outside the viewport are sent in a low quality. As long as no interaction happens with the ODV, the user always sees a high quality video. However, when a user pans, he will possibly see lower quality tiles for a short period of time before tile quality is upgraded. A user study involving 14 participants revealed that, users generally like this technique when the difference between the lower quality and high quality is not too big. However, when a very low quality is used, users still like the technique if the quality switches fast enough. When low quality tiles were visible for a longer period of time, people generally did not like this technique. The user study also showed that the user’s annoyance of the low
quality can vary depending on the video content. In the second solution, we used multiple layers of quality to represent the video. In this case, we determine the high quality parts of the video in a preprocessing step. Typically, these high quality parts correspond with a foreground layer. Multiple layers have to be reconstructed to account for the resulting video. To recompose these layers, we used a threshold filter on black or bright green colors. Both chroma keys have their optimal use scenarios. Black is often located in video footage, therefore, making it impossible to use high thresholds. Bright green is mostly not present in real-life video, therefore, we can use higher thresholds. However, when only a small portion of the green chroma key remains in the video, it is clearly visible. Even if some black pixels remain in the video, it does not attract that much attention as a green color would. For both solutions we created proof-of-concept demos to showcase the functionality.

When watching a video, we often watch it on one screen. However when using ODVs we only see a part of the video at all times. To enlarge the portion of the video footage we can see, we created a solution that utilizes multiple screens to render the viewport on. The proposed solution uses a master-slave principle to minimize the amount of data being sent over the Internet. Slaves are only used to display video footage, while the master has to forward the DASH data. We can conclude that the master actually has to do some added work. However, the benefit of this solution is that the amount of work does not scale with the amount of slaves.

All these use cases are part of a case study to determine how we can realize advanced use cases using MPEG DASH. They each showcase their own unique functionality. The approaches are general enough to incorporate with other video streaming techniques. Therefore, it also enables us to use the different solutions created in each use case simultaneously. We created two techniques to save bandwidth, which can be used in a non-linear video scenario. The non-linear video can in its turn be displayed on multiple screens. Even though, they can work together, these solution can be used in context of other work based on video streaming.

Because these use cases were created on an implementation framework that is currently under development, some of the possible future work is related to improving this framework. First and foremost, the lack of a decent SegmentDownloadRateEnforcer has to be addressed. More quality adaptation and precaching logics can be created to account for different functionality. Improvements can be made to each use case to have additional or improved functionality. The current solution for the non-linear video has trouble playing high resolution videos. A possible solution is the use of multi-layering or tiling. However, we also found that a large portion of the work of the application is decoding Base64 messages. In the future, a scheduler can be made to manage the work done by decoding the message when no other important operation has priority. In the multi-screen use case, it can be interesting to look at using the screens of smartphones or tablets and thus adjusting the data to the location of the device in the setup. When playing multi-layered videos, we could get a better result when a more advanced reconstruction algorithm would be used. This algorithm should take care of any artifacts that are present in our current implementation. In the tiled approach, we could improve the quality adaptation logic to take the available bandwidth into account. During the user tests, a few participants noted that smoothing on the edges would probably improve the result drasti-
cally. The visual fidelity of the tiling approach could possibly be improved by defining regions that contain an object to focus on. By sending the tiles of this focus object in a decent quality, users can focus on this object after a panning motion. This means that they will not experience the low quality tiles as they would without available focus. Finally, we worked exclusively with on-demand video content. As future work, we can look at similar functionality in the case of live video content.

To summarize we studied multiple advanced video streaming concepts in this thesis and thereby made several contributions:

- we created a web-based solution that can provide a personalized non-linear video
- the switching time between sequences in the non-linear video is low enough to provide seamless switching
- the user study has shown that when a transition is short enough, it does not influence whether a user accepts the result
- we created a tiling approach that maximizes the quality of the viewport of an ODV, while compromising the quality of the rest of the video.
- users like the tiling approach when the switching time is low (i.e., 2 or 4 seconds), when not too much low quality video footage is visible and when a medium quality is used outside of the viewport.
- we created a multi-layer solution that is able to reconstruct multiple depth layers of video using both a black or a bright green chroma key.
- we contributed to a multi-screen solution that enlarges the viewport of an ODV, which we created the framework and communication library for

We have shown that MPEG-DASH lends itself well to the creation of advanced video streaming use cases. We focused on a small subset of possible applications and implemented specific functionality for this. Because these use cases go beyond normal video playback, people have to get familiar with this technology. When the concepts in this thesis are more well known by the general public, I expect that the techniques can be widely used. There is an initial learning curve, but in my opinion the techniques prove to have added value. The ideas behind tiling and multi-layering are only usable when a user knows he has to compromise the overall quality of the video. In my opinion when a user is aware of this, both techniques are acceptable solutions to the bandwidth problem of high resolution videos. Some people already use multiple computer screens to enlarge the amount of displayed data. The same idea can be used in the context of video streaming. Therefore, I find the multi-screen solution very useful when a user does not want to compromise video details but wants to view a larger portion of the video.
Bibliography


Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:

Applying MPEG DASH in the browser for non-traditional video streaming scenarios

Richting: master in de informatica-multimedia
Jaar: 2015

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen -, aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtredt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Beertens, Geoffrey

Datum: 22/01/2015