Viewport dependent MPEG-DASH streaming of 360 degree natively tiled HEVC video in web browser context

Dissertation presented to obtain the degree of master in computer science/ICT/knowledge technology

Moesen Gert

Academic year: 2017-2018

Supervisor:
Prof. dr. Quax Peter

Co-supervisor:
Prof. dr. Quax Peter

Mentor:
Dr. Wijnants Maarten

Abstract

H.264/MPEG-4 AVC has been the mainstream video consumption standard for years. However, with the current diversity of services, the still growing popularity of high quality videos, and continuously increasing video resolution and qualities, a more efficient coding performance is demanded than H.264/MPEG-4 AVC can supply. Therefor, new video codecs are being developed to support these high resolution videos. One of those video standards is H.265/HEVC which supports for the same quality videos up to 50% decrease in size compared to H.264. Not only did HEVC have a huge impact on the files size, it also came with native support of tiles. Tiles allow a video to be divided in rectangular regions. These regions can then be coded independently of each other.

With the announcement of HEVC support in the Apple Safari and the Microsoft Edge web browser, we wondered if it was possible to natively stream tiled HEVC MPEG-DASH content and decode this in web browser context. By using MPEG-DASH, an adaptive control of the quality per tile can be gained. This is done by using an AdaptationSet per tile and allowing the tile to have multiple representation qualities. The adaptive control is needed as we work with 360 degree video in which a user has only a spatially limited viewport. Based on the viewport and its position in regard to the tiles, we can adjust the quality of the tiles.

We started by asking ourselves if the HEVC tiled MPEG-DASH stream can be reformed to a native tiled HEVC video in the web browser. The restructure of the tile stream is needed to play the HEVC video content. This is done by remapping the input streams for each tile into one native tiled video bit stream which the HEVC decoder can then decode. By doing this, video playback is possible. In this thesis, we test and implement the restructure process in a web application and measure the preprocessing times to show if live playback is feasible.

Acknowledgements

Firstly, I would like to thank my mentor dr. Maarten Wijnants for his continues help and feedback throughout this thesis. I would also like to thank my supervisor prof. dr. Peter Quax and co-supervisor prof dr. Wim Lamotte for their guidance.

Secondly, I would like to thank my family for their continuous support throughout the years. And also my friends on whom I could always count.

Contents

Acronyms

CABAC Context-Based Adaptive Binary Arithmetic Coding. 3

CB Coding Block. 6, 7

CDN Content Delivery Network. 16

CTB Coding Tree Block. 6, 7, 33

CTU Coding Tree Unit. 4–9, 11

CU Coding Unit. 6–9

HEVC High Efficiency Video Coding. xi, 1–13, 26, 29–36, 43, 45, 61, 62

MPD Media Presentation Description. 16–19, 21–23, 33, 34, 46, 47

MPEG Moving Picture Experts Group. 16

MPEG-DASH Dynamic Adaptive Streaming over HTTP. 2, 16, 23, 25–27, 29–32, 34, 35, 39–41, 49, 61

MTU Maximum Transmission Unit. 3, 4

NAL Network Abstraction Layer. 9–13

PB Prediction Block. 6, 7

PPS Picture Parameter Set. 11–13, 33

PU Prediction Unit. 6–9

SEI Supplemental Enhancement Information. 11–13

SPS Sequence Parameter Set. 7, 11–13, 33

SRD Spatial Relationship Description. 17, 21–23, 27, 34

TB Transform Block. 7

TU Transform Unit. 6–9

URL Uniform Resource Locator. 16, 18, 19

VCL Video Coding Layer. 9, 11, 12

VPS Video Parameter Set. 11, 12, 33

VR Virtual Reality. 25, 29

VUI Video Usability Information. 13

WPP Wavefront Parallel Processing. 6

Listings

List of Figures

Chapter 1

Introduction

H.264/MPEG-4 AVC has been the mainstream video consumption standard for years. However, the current diversity of services, the still growing popularity of high quality videos, and continuously increasing video resolution and qualities are demanding a more efficient coding capability than H.264/MPEG-4 AVC can supply [SO10] [SOHW12]. Moreover, mobile devices are generating more and more download traffic over wireless networks which are more prone to transport errors and have less throughput than their wired counterparts. These mobile networks are in no way optimized for streaming the high quality videos requested. Even though more bandwidth is available every day, the need for better compression of videos with higher-resolution content is higher than ever before to feed the ever raising demand for video content over the Internet [SCF+12]. As a result H.265/HEVC has been designed in a joined effort of ITU-T VCEG and ISO/IEC MPEG standardization organizations to improve on the shortcomings of H.264/AVC by supporting higher resolutions and by decreasing the bitrate requirement of coded videos. This is done by allowing for size reductions of up to 50% for videos at comparable perceptual qualities [CAMJ+12].

A more efficient encoding comes with a higher computational power needed to encode and decode HEVC videos. While single-core processors can now easily decode a 1080p H.264/AVC video in real-time, it is still unsure if a single-core processor can decode 2160p HEVC video in real-time without hardware decoding. Luckily, HEVC hardware decoding is gaining ground over the last few years and becoming more popular every day. Even though multiprocessor devices became more and more the mainstream, H.264/AVC was developed without parallelism in mind; H.264/AVC jumped on the bus by allowing parallelism via *slices*. However, parallelism introduced via *slices* was an afterthought and in desperate need to be implemented more efficiently in the new HEVC codec. The HEVC codec was designed with parallelism in mind. With this mindset, real-time decoding was in grasp on multi-core CPUs thanks to parallelism despite the increased processing power needed. HEVC supports a few approaches to parallelize the decoding which will be fully explained in Chapter 2. One of those is *tiles*. *Tiles* allow a video to be divided in rectangular regions. These regions can then be decoded independently of each other. *Tiles* have been used to adaptively stream omnidirectional content in H.264/AVC which will be discussed in Chapter 4. By using *tiles* a video can be cut in similar size rectangles. By interchanging these *tiles* with a different quality, a video based on the available qualities can be presented.

Presenting the video via Internet can be done in different manners. The one used in this thesis is Dynamic Adaptive Streaming over HTTP (MPEG-DASH) which is explained in Chapter 3. MPEG-DASH allows the streaming client to adaptively change the quality of the video by interchanging video segments. A DASH-Client can choose to change quality because of fluctuating bandwidth capacity, or a user setting change. The different qualities we can choose from are defined in the **AdaptationSets**. Each **AdaptationSet** contains one tile with the different possible **Representations** from which we can pick the desired quality. There is also one **AdaptationSet** which contains only the non video coding layer NAL units which are needed to create the native tiled video. By combining all these **AdaptationSets** in the correct order, it is possible to create a single natively tiled video bit stream which can be decoded by the web browser. NAL units will be fully explained in Chapter 2. In this thesis the quality will be decided based upon the viewport of a user. By allowing a user to look around in the 360 degree video, the quality will change based on the *tiles* that intersect with the current viewport. The full explanation of the implementation is given in Chapter 5.

In this thesis, we will focus on how a native tiled HEVC video can be played in the web browser. Therefor, the main questions will be:

- Is preprocessing a MPEG-DASH tiled video in a web browser to a native tiled HEVC video feasible?
- Can the preprocessing be implemented in a sufficiently efficient manner to ensure live playback?

Thereby defining this thesis as a feasibility study.

These question were verified in 1920x1080 (1080p) and 3840x2160 (2160p) resolution videos. As the segment duration can be chosen freely by the content creator, two segment durations were tested, more precisely 1000 and 5000 milliseconds, to measure the impact on the preprocessing step. These results will be discussed in Chapter 6. In Chapter 7 the conclusion of this master thesis will be given, together with possible starting points for future work.

Chapter 2

H.265/HEVC

H.265/HEVC has been designed in a joined effort of ITU-T VCEG and ISO/IEC MPEG standardization organizations to improve on the shortcomings of H.264/AVC by supporting higher resolutions and by decreasing the bitrate requirement of encoded videos. This is done by realizing a file size reduction up to 50% while preserving the same quality [CAMJ+12].

HEVC supports several approaches to parallelize the decoding and encoding which will be fully explained in this chapter. One of these approaches is the use of *tiles*. *Tiles* allow a user to divide a video in rectangular regions that can be individually encoded and decoded.

As the encoding and decoding of the H.265 video codec is not the subject of this thesis, it will not be described in detail. A detailed explanation of the complete workings of HEVC can be found in [SOHW12]. This chapter will give an overview of HEVC and some of the features needed in this thesis for the implementation of tiles support in browsers.

2.1 Slices

Slices were used for parallelization in H.264/AVC even though this was not the original purpose of slices. Originally, slices were designed to map video bit streams into smaller chunks for transmission over the Internet. Hereby, could the size of a coded slice be determined by the network characteristic, take for example the Maximum Transmission Unit (MTU) size of a network. If the coded slice is smaller or equal to the MTU, a complete slice can be sent in just one package [MSH+13]. However, this can decrease the coding efficiency as the MTU size does not take in consideration correlations between pixels.

Even thought slices where originally developed for resynchronization in case of packet loss or bit stream errors, they have been progressively used for parallelization as slices allow for reconstruction of parts of the video picture independently of other slices. However, there are a few negative effects of using slices for parallelization. One of those is the effect of the coding efficiency reduction by breaking the Context-Based Adaptive Binary Arithmetic Coding (CABAC) prediction dependencies at the slice boundaries. CABAC is one of two entropy encoding standard used in H.264/AVC. More information about CABAC can be found in [MSW03]. Another negative effect of slices is the parsing overhead introduced to

find each each slice entry point at decoder side. Furthermore, for slices to be decodable, they need header info, which further decrease the coding efficiency by adding the header info with each slice or waiting for the slice with the header info before decoding can start. There can also be load balancing issues because of the diversity in slice size and complexities [GNA14].

Within HEVC, there can be a few options on how to pick slices. For example, based on network constraints such as MTU, or the amount of pixels that can be processed by limiting the amount of CTUs. Another option is to define a slice per tile. A more detailed explanation of tiles and their constraints will follow in Section 2.2.

When slices get processed, it will be in raster scan order. A visual representation of the raster scan order constrained by tiles can be seen in Figure 2.1. Notice that the sizes of a tile is the same as the size of one slice in Figure 2.1. This does not necessarily have to be so.

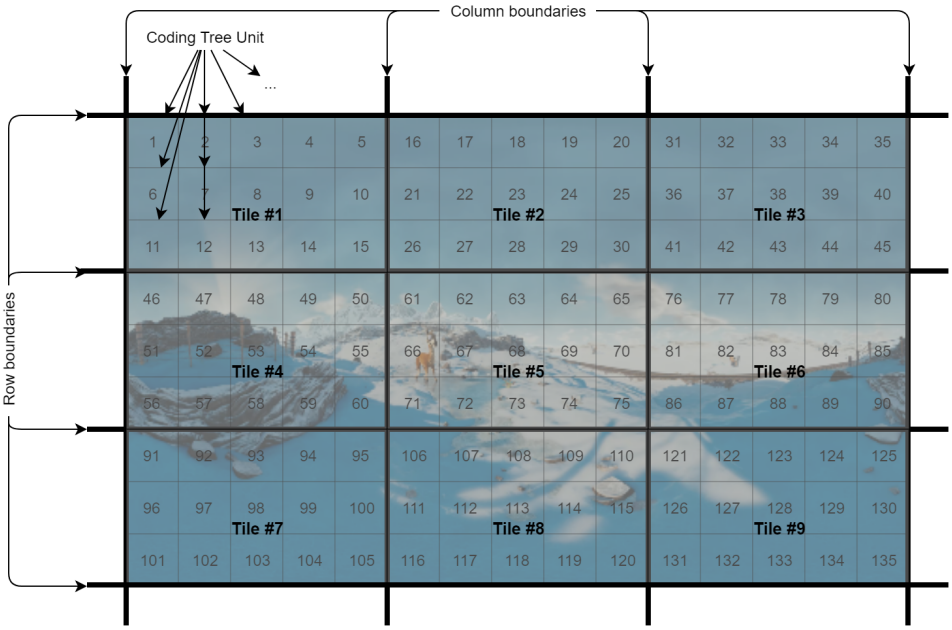


Figure 2.1: A frame out of a video. The picture is divided in a 3x3 tiled picture divided by the CTU boundaries

2.2 Tiles

Tiles is one of the most popular new features of High Efficiency Video Coding (HEVC). With tiles, a video frame can be divided into multiple independent rectangular regions. More precisely, these tile regions increase the parallelization by allowing a CPU to process each region independently. Furthermore, coding efficiency is drastically increased in comparison to the older slice-based methods were flexible macroblock ordering (FMO) in slices was allowed to enable slices of any size. This is because slice headers are not needed for tiles unlike FMO [MSH+13]. Not only does parallelization increase, there are more positive features about tiles which will be discussed throughout this section.

By not needing a tile header, the coding efficiency is increased; the same picture region

can be encoded in less bits than when slices were used. But this does bring the question of how a tile is defined. A tile is defined by x horizontal and y vertical amount of Coding Tree Unit (CTU) (see Section 2.4 for more info about CTU). Logically, this comes with the consequence that tiles are link to CTU boundaries and do not allow half CTUs. A visual representation is shown in Figure 2.1. A single video frame in HEVC context is partitioned into CTUs.

All tiles will be processed in horizontal raster scan order. This means, by looking at Figure 2.1, first tile #1, tile #2 till tile #9. Within the tiles, the encompassed CTUs will be again processed in raster scan order. Notice how the last CTU of the first tile is 15 and the first CTU of tile #2 is 16. By processing the CTUs within a tile in scan order with this restricted tile pattern, the needed on-chip memory is reduced. This is because of the reduced line buffer requirements for motion estimation. Following equation shows the needed data storage for a situation without tiles [MSH+13];

$$data\ storage(without\ tiles) = \texttt{PicW} * (2 * \texttt{SRy} + \texttt{CTUHeight})$$

- `PicW` represents the width of the picture.
- `SRy` represents the maximal vertical size of the motion vector
- `CTUHeight` the height of the CTU.

As one can imagine, the data storage can get extremely big in situations with content resolutions up to 4k. When the `PicW` is a lot bigger than the `TileW`, the saved data storage space can be substantial. The following equation shows the data storage calculation for tiled pictures.

$$data\ storage(with\ tiles) = (\texttt{TileW} + 2 * \texttt{SRx}) * (2 * \texttt{SRy} + \texttt{CTUHeight})$$

- `TileW` represents the width of the tile.
- `SRx` represents the maximal horizontal size of the motion vector
- `SRy` represents the the maximal vertical size of the motion vector
- `CTUHeight` the height of the CTU.

The location of the tiles must be signaled in some sort in the bit stream to make sure decoding is possible. This is done by writing the offsetting in the slice header for all but the first tile. The reason for not writing the first tile in the header is because the first tile immediately follows the slice header, which is know at the decoder.

2.2.1 Constraints on tiles

A restriction on tiles is that their allowed size is bound by the CTUs. A tile width has to be equal or greater than 256 luma samples, which are 4 CTUs of 64 pixels. The minimum height is 64 luma samples, which is one 64 pixels CTU, two 32 pixels CTUs or four 16 pixel CTUs. With this constraint, it is made sure that tiles are not too small. The maximum amount of tiles is also constrained based on the bit stream level under consideration [MSH+13].

Furthermore, constraints are enforced upon the combination of slices and tiles. This is so the decoder complexity is manageable. Either all CTUs in a slice belong to the same tile, or all CTUs in a tile belong to the same slice [IT18]. It is possible to allow more than one tile per slice and more slices in one tile per picture.

Another constrained in this thesis to allow for parallel decoding of the tiles is the motion vector restriction. The motion vectors will be limited to CTU within the same tile. This means that as an object moves over tile boundaries, artifacts can be shown as the motion prediction is restricted within a tile.

2.3 Wavefront Parallel Processing

Another parallelization method developed for HEVC is Wavefront Parallel Processing (WPP). WPP has the upper hand when it comes to achieving higher compression and less to no visual artifacts when comparing to tiles [GNA14] [SOHW12].

WPP achieves this by splitting the slices in CTU rows. The full explanation of a CTU can be found Section 2.4. When the encoder or decoder starts encoding or decoding the video picture, it depends on the previously processed row. Parallelization is gained by allowing to start processing the current row as soon as two CTUs in the previous row are processed. Add this to the fact that each row is processed in raster scan order, meaning that the picture gets decoded from top-left to right-bottom.

WPP does come with the drawback that there needs to be communication between the different processing cores as information of the previous CTU row is needed to decode the current one. A visual representation is shown in Figure 2.2.

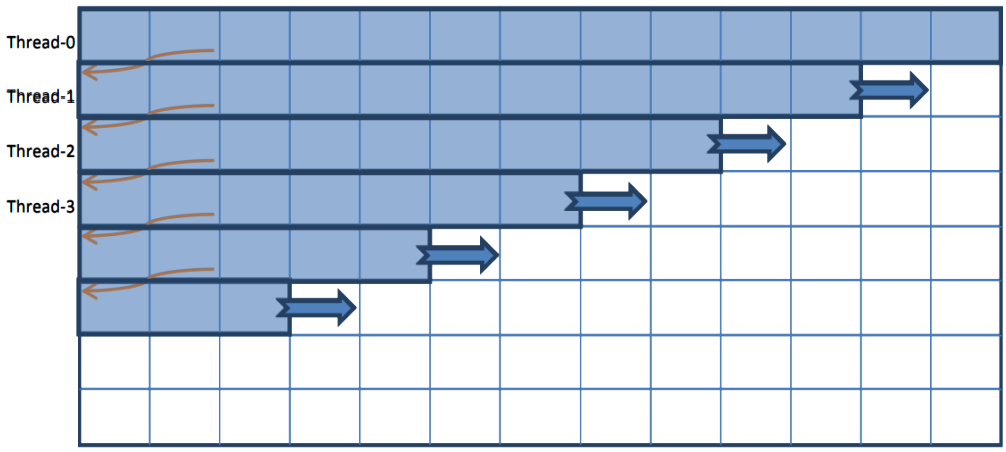


Figure 2.2: Visual representation of Wavefront Parallel Processing [GNA14]

2.4 Coding Tree Unit

Before any more explanation about HEVC can be done, a few important acronyms have to be introduced; Coding Tree Unit (CTU), Coding Unit (CU), Prediction Unit (PU), Transform Unit (TU), Coding Tree Block (CTB), Coding Block (CB), Prediction Block

(PB) and Transform Block (TB). As mentioned before, HEVC is more efficient in compression of video data in higher resolutions. One of those reasons is because of the usage of a new structure with CTU instead of macroblocks.

The best way to start dissecting the HEVC standard is to look at the naming convention. Take for example Coding Tree Unit and Coding Tree Block. Notice how they both start with *Coding Tree* and differ in *Unit* and *Block*. When a name ends in *Unit*, a coding logical unit which in time will be encoded into a HEVC bit stream is meant. However, when something ends in *Block*, it indicates a process that targets a portion of the video frame buffer [Mot12]. More precise, the terms; CTB, CB, PB and TB defined specify the 2-D sample array of a color component. For example, CTU consists of one luma CTB, two chroma CTBs and other syntax elements related to the CTU [KML+12]. A visual representation can be seen in figure 2.3. The same strategy can be carried on to CU, PU and TU. Another uncommon possibility for representing a CTU that will not be further discussed is when CTU is a CTB of a monochrome picture or a picture encoded using three color planes [IT18]. However, in this thesis we look at 3 CTBs per CTU.

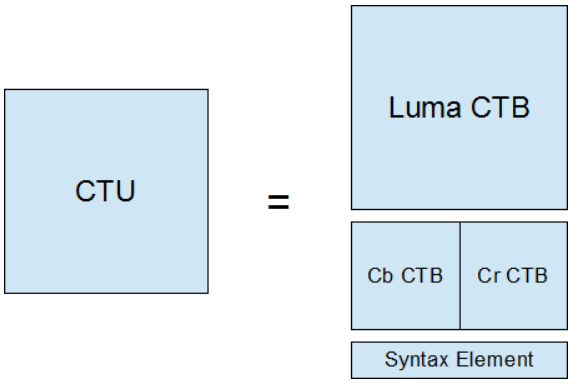
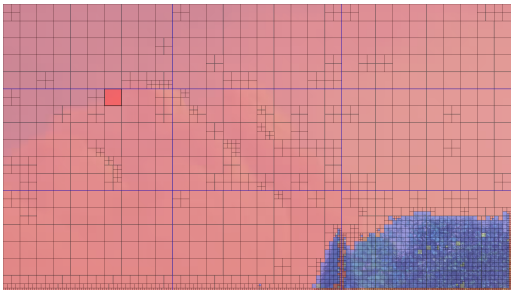


Figure 2.3: CTU to CTB [Mot12]

Now that the relationship between a CTU and CTB is know, the explanation will continue with the use of CTU. At the top level of a HEVC picture (a single frame of a video) there are Coding Tree Unit (CTU). The easiest way to explain these is to look at CTU as the equivalent of a macroblock in H.264/AVC content [Mot12]. One of the most invasive changes between macroblocks in H.264/AVC and CTU in H.265/HEVC is the ability to change the CTU dimensions in context of the HEVC video. This is in contrast with the fixed size of the 16x16 macroblock [KML+12]. In HEVC a CTU can have a size of 64x64, 32x32, 16x16 pixels. The size of the CTU is the same over the whole video sequence. Furthermore, this has a direct influence over the coding efficiency of HEVC.

As the CTU can change depending on the video, it needs to be defined. The width and height of the CTU are defined in the Sequence Parameter Set (SPS). A more detailed view of the SPS is given in Section 2.5.1. As one can imagine is a CTU size of 64x64 or 32x32 pixels in most situations too big to know if inter-picture prediction or intra-picture prediction should happen; for example, when their are a lot of details like snow falling in a frame. For this reason, the CTU can get subdivided in multiple CUs of smaller sizes. The size of a CU can go from 64x64 down to 8x8 pixels depending on the definition of the maximum and minimum CU size in the SPS. This subdividing is defined by a Coding Tree and is called CTU partitioning. The next steps are also part of the CTU partitioning. A visual example of the CTU partitioning can be seen in Figure 2.6. By allowing such a dynamic use of CUs, a video can be encoded with less symbols as a bigger CU can

(a) A picture from a HEVC video where the tiles are shown.



(b) The same frame with the CTU overlay with pixel size 32x32 and subdivided CU (TU and PU) ranging pixel size 4x4 to the same size as the CTU (32x32 pixel size)

Figure 2.4: Still of a video with overlaying CTU

be chosen compared to the H.264/AVC context with a fixed size of 16x16 pixels for the macroblock.

Once a CTU is subdivided into CUs, the choice between inter- or intra- prediction can be made as 8x8 pixels is small enough in most cases. Take for example Figure 2.4. In this image, there is a big sky with a forest emerging on the right bottom (excerpt from Big Buck Bunny [GR08]). For the flat surfaces such as the white cloud centers, a CTU of 64x64 or 32x32 is used depending on the video size. In this example, a CTU size of 32x32 pixels has been chosen as the video is only in a full HD resolution (1920x1080 pixels). When more details emerge as with the forest and the edges of the cloud, the CU size of 32x32 pixels will be too large to determine inter- or intra-image prediction. By subdividing into smaller CUs of 16x16 or even 8x8, more precise decisions on inter- or intra prediction can be made. For most situations this is enough but for the situation where tree tops are shown, even 8x8 pixels is not fine enough to get a correct match. The tree top might fill half the 8x8 CU, so different motion vectors for the same CU might be interesting.

A CU can be split into one, two or four PUs depending on the PU splitting type and the CU prediction mode. This allows for a predictions per PU. Equal to previous standards, one of three CU prediction categories is used:

- *inter coded CU*, makes use of motion compensation schemes to predict the current block
- *intra coded CU*, makes use of nearby reconstructed samples in the same frame.
- *skipped CU*, is a special case where the combination of motion vector difference and residual energy equals to zero [KML+12].

For each category, the Prediction Unit (PU) splitting type is different as can be seen in Figure 2.5. In the HEVC context, there are two splitting types for intra coded CU, eight for inter coded CU and only one for skipped type [KML+12].

Similar as for PUs, multiple TUs are possible per CU. The TU contains residual or transform coefficients for integer transformation of quantization by the decoder. More details about the TU can be found in [KML+12].

As described before, a CTU partitioning happens recursively and can best be seen in the

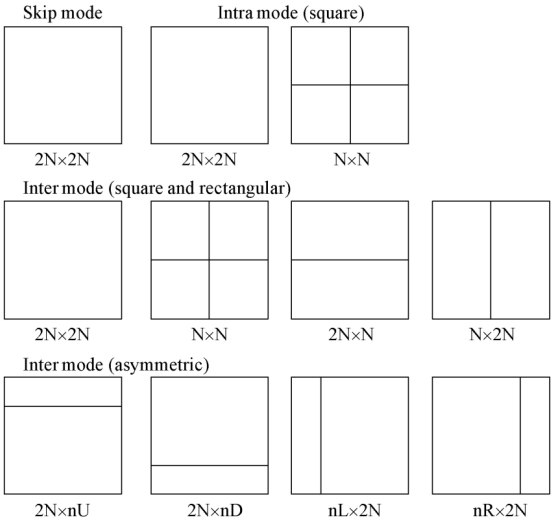


Figure 2.5: Prediction Unit splitting types [KML+12]

example in Figure 2.6. In this image the generated *Coding Tree Unit* is show on the left, while on the right the visualization on the video picture for the CTU with corresponding CU is given.

The processing of the CTU to generate the CU starts in raster line order. This means that the top left with the number one is the first to start. In this example the CTU has to be split up for the CU of a size of 32x32. This has the consequence that in this CTU all CUs have to be of size 32x32 or smaller as distribution has to be of dimension NxN. If no further split-up is needed, next CU is processed till all CUs are done.

Take for example the first CU, in Figure 2.6. When processing this video, no further split up has to be done as there are no real details in the image. This means the PU en TU have to be chosen, these are represented by the nodes. The next target following the raster scan order is taken, number 2 in the figure. Only one PU is needed which is represented in the node, but 4 TUs are calculated as the match is not good enough for the whole 32x32 CU. For each 16x16 PU values are calculated which end in a node. These steps are continued for the complete tree.

2.5 NAL Unit

Network Abstraction Layer (NAL) encapsulate the coded video data so that it can be stored [Wie14]. Within the NAL unit there are two types of data: Video Coding Layer (VCL) and non Video Coding Layer (VCL). The VCL contains all slices and CTU data. The non VCL data consists of all other information. More details will be given later on in this section.

A Network Abstraction Layer (NAL) unit consists of a NAL unit header followed by a NAL unit payload. To mark the end of the payload a stop bit 1 is used, followed by one or more 0's for byte alignment.

In the introduction of Chapter 2 it was mentioned that the development team of H.265/HEVC chose to also include NAL unit headers, taken from H.264/AVC. However, HEVC de-

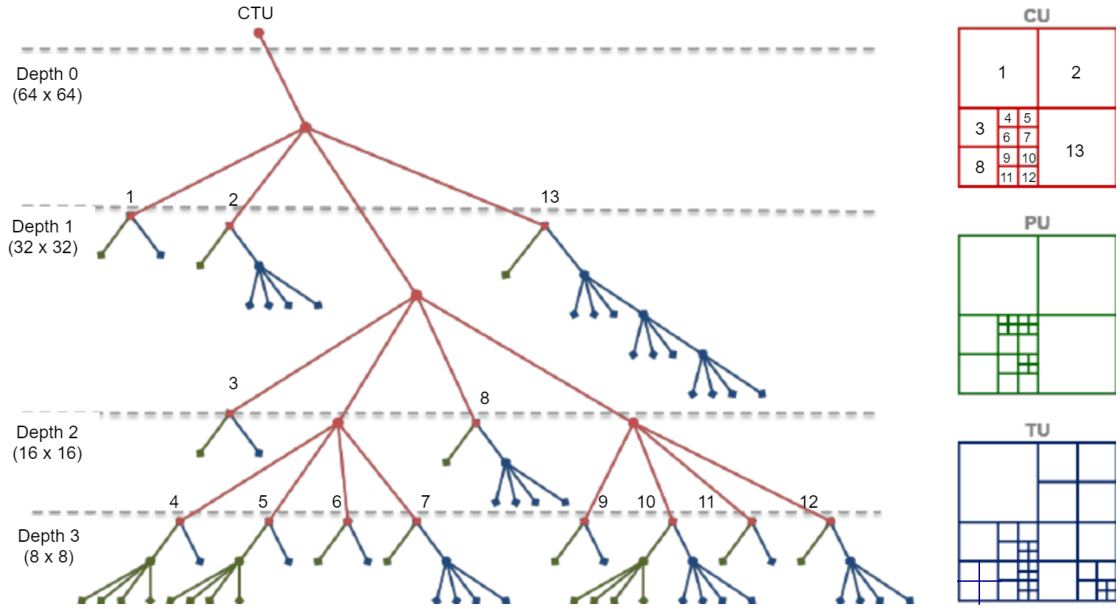
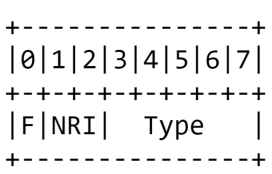
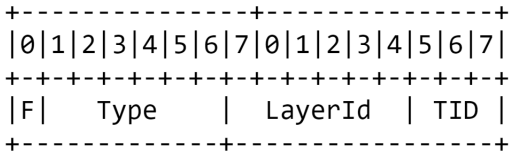


Figure 2.6: CTU partitioning hierarchy example with corresponding CU, PU and TU. Inspired by [RCAFE+14]

fines a two-byte NAL unit header [SCF+12] instead of the one-byte NAL unit header of H.264/AVC. The NAL unit header can be seen in Figure 2.7 where Figure 2.7a shows the one-byte header in the H.264/AVC standard [SWH+05] and Figure 2.7b the two-byte NAL header according to RFC 7798 [WSS+16]. The developers chose for a two-byte NAL unit header as this would make the header more future proof for the introduction of HEVC scalable coding and 3-D video encoding.



(a) One-byte NAL unit header of H.264/AVC



(b) Two-byte NAL unit header of H.265/HEVC

Figure 2.7: NAL unit headers for the H.264/AVC and H.265/HEVC video codec

NRI of the H.264 NAL unit header has not been adopted in HEVC, as can be seen in the Figure 2.7. *NRI* is a two-bit codeword that would allow for data partitioning. As HEVC does not allow partitioning for sending the NAL unit in parts over a network, this field was unneeded. The structure of the H.265/HEVC NAL unit is as follows:

- *F* consists of 1 bit and stands for the *forbidden_zero* field. This field has to be set to 0, if not the assumption might be that the NAL header is faulty by syntax violation and should be discarded or requested again. The reason for including this *forbidden_zero* in the NAL unit header is to enable the transport of HEVC video over MPEG-2 transport systems, by including it the start code emulations in MPEG-2 legacy environments are prevented [WSS+16].
- *Type* consists of 6 bits and, just as the name suggests, represents the NAL unit type. Because the *NRI* field is unneeded in HEVC, the HEVC NAL unit field *Type* is one

bit bigger compared to H.264/AVC, extending it to 64 unique values to define types. The other unused bit of the H.264 *NRI* field is used for possible future extensions. If the most significant bit of the *Type* field is equal to 0, the NAL unit defines a VCL-NAL unit [SCF+12]. Detailed explanation of Video Coding Layer (VCL) can be found in [SHWW12]. However it is important to understand that the VCL contains the coded video sequence data, this can be a slices and CTUs. Non-VCL NAL units contain control information or parameter sets that mostly applies to multiple coded video frames.

- *LayerId* consists of 6 bits and is required for the layer identifier. In the first version of the HEVC specification, this field had to be 0. Upward from the first version, this identifier is used for the HEVC extension for scalable video coding and multiview coding [Wie14].
- *TID* consists of 3 bits and represents the *TemporalId*. It is unauthorized to have a *TID* value of 0, this is to ensure that at least one bit in the NAL unit header is set. Because of the 1 a differiantiation can be made between the header of the NAL and the NAL unit payload. The *TemporalId* will be the id set in the *TID* field minus one [WSS+16].

By having a clear definition of what the NAL unit is, a few common types of NAL units will be discussed. For a full detailed view of all NAL units and their meaning: [ISO17].

Figure 2.8 shows part of a HEVC bit stream. This bit stream is taken from a 3x3 tiled HEVC video. To analyze a HEVC bit stream, the tool *Zond 265* has been used [Mul18].

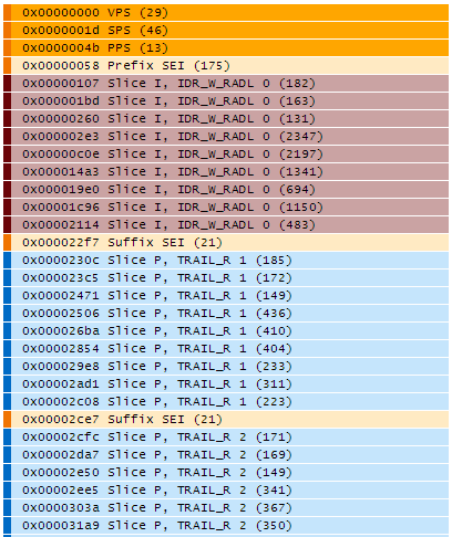


Figure 2.8: Bit stream of a HEVC video

In Figure 2.8, the HEVC bit stream starts with 4 non VCL NAL units, each representing another type: Video Parameter Set (VPS), Sequence Parameter Set (SPS), Picture Parameter Set (PPS) and Supplemental Enhancement Information (SEI). VPS, SPS and PPS are three types of *Parameter Sets*. More details of the *Parameter Sets* will given in Section 2.5.1. It is sufficient to understand that this is a hierarchical structure that gives information to decode the HEVC video. For the video bit stream sample shown in the figure, only one VPS, SPS and PPS are given. All slices will reference to these non VCL

NAL units to decode the whole video. A more detailed look on slices and tiles has been given in Sections 2.1 and 2.2, respectively. In Figure 2.8, each Slice represents one tile and can be decoded independently.

2.5.1 Parameter Set

As shown in Figure 2.8, the bit stream starts by defining the Video Parameter Set (VPS), Sequence Parameter Set (SPS) and Picture Parameter Set (PPS). These parameter sets follow a hierarchical structure where the PPS refers to the parent SPS and the SPS in turn refers to the parent VPS. A visualization of this structure can be seen in figure 2.9.

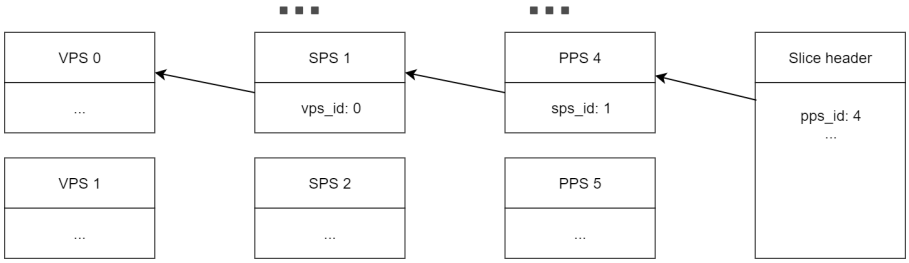


Figure 2.9: Example Parameter Sets referring to each other in a hierarchical structure [SCF+12]

When talking about parameter sets, there are two options, either they are part of the bit stream or delivered and stored separately. Just as a lot of other features of HEVC, parameter sets are an offspring of the already existing parameter set features in H.264/AVC. However, new features were added to keep up with the time and to overcome some of the hold-backs of the older generation codec [SCF+12].

Originally, parameter sets were developed in response to the effects of losing a picute or sequence header when a video frame is partitioned into segments prior to network transmission. Take for example the scenario where a video frame is partitioned into slices, see Section 2.1. All these slices are transported on a medium that can produce errors or, in the worst case, lose a slice. When the first slice would get lost, which contains the picture header with all info to decode the picture, a picture can become undecodable and has to be dropped. This is where parameter sets step in, as shown in Figure 2.9, the slice header reference the PPS, the PPS will reference the SPS and the SPS will point to the VPS. In example Figure 2.8 only one VPS, SPS and PPS is given for the complete video bit stream. All slices will refer to these non VCL NAL units to decode the whole video.

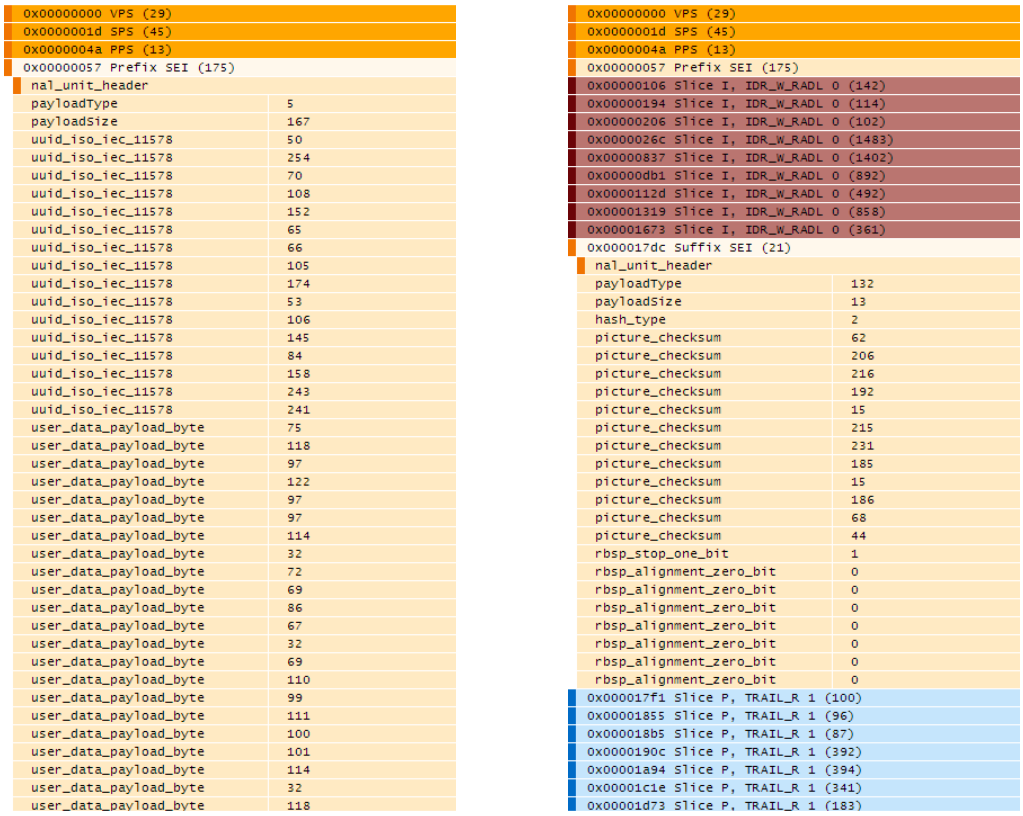
At the top of the hierarchy there is the Video Parameter Set (VPS). This parameter set is new compared to H.264/AVC and has been introduced for HEVC to work on the shortcomings of resending the same info in the SEI and SPS (this will be explained in Section 2.5.2). The VPS represents the information of all layers in the bit stream and eases the use of Scalable Video Coding (SVC) and Multiview Video Coding (MVC) extensions [Sul]. Next is the SPS, this parameter set applies to the entire video sequence. A coded video sequence is series of sequential access units of which all contain to the same NAL unit stream and are linked to the same SPS. Within the SPS, there is a field that stores the identifier for the associated VPS. Other SPS fields are the description of the

usage of coding tools and their parameters, or for example Video Usability Information (VUI) with information that has no direct impact on the decoding process.

Lastly there is the PPS with the ability to change for different pictures of the same coded video sequence. However, as seen in Figure 2.8, slices can refer to the same PPS. Just as in the SPS, the PPS has an identifier to the parent (SPS in this case) in the hierarchy. The parameters of the PPS other than the SPS identifier, describe the coding tools that should be used in the slice that refers to the PPS. Also the parameters for the coding tool are included.

2.5.2 SEI

Another NAL unit that is common in the given bit stream of Figure 2.8 is Supplemental Enhancement Information (SEI). In H.264/AVC there was already such thing as a SEI. However, in HEVC they chose to extend upon the predecessor SEI message by allowing two types: prefix SEI message, which already existed in H.264/AVC and a suffix SEI message, introduced in HEVC. Either type of SEI message provides meta data and is not required for the decoding process [Wie14]. Some SEI messages can also be either prefix or suffix, the payload will be the same. SEI units are used to make the decoder determine certain features that otherwise might be computationally hard to determine and such use extra time to decode the video bit stream. Figure 2.10 shows a prefix SEI and suffix SEI NAL unit, the suffix SEI message contains the type 132, meaning the payload had checksums to make sure that the decoded picture matches that produced by the encoder. The prefix SEI message contains SEI type 5 and stands for *User Data Unregistered*, this means that the unregistered user data identified by a UUID in this message is free to have unspecified contents according to [ISO14a].



(a) Prefix SEI message **(b)** Suffix SEI message

Figure 2.10: SEI NAL units for a H.265/HEVC video bit stream

Chapter 3

MPEG-DASH

3.1 Why MPEG-DASH?

One of the first well-known video and audio content delivery techniques over the Internet was RTP, shorthand for Real-Time Transport Protocol. RTP is a UDP-based streaming protocol which maintains a streaming context at serverside. When development of RTP started, the main goal was to deliver large amounts of video and audio data over the Internet. However, the first problem with RTP streaming is RTP package blocking by the firewall. There are also security risks when when a RTP package is allowed by the firewall by making use UDP packets for transport. Firewalls have difficulties in filtering out malicious data injected into a RTP stream [LSNHS05]. A second issue is the scalability for RTP streaming sessions. When a RTP streaming session gets started by the client, the server needs to save information on the current session. Without this information, the server is unable to send real-time video and audio content to the expecting client. This kind of communication also comes with a overhead as there needs to be an initial exchange to start communication between the server and the client. These two big issues ask for a better way of sending video and audio content over the Internet [Sod11]. Another possibility to streaming is progressive downloading of the media by requesting a byte range of the media file over HTTP. However, the HTTP web server has to support byte range requests and a few more disadvantages [Sto11]. Take for example when an user stops watching or listening to the media and the progressive download has started, this is a waste of bandwidth. Another disadvantage is not supporting live media content which is becoming more and more important.

The growth of the World Wide Web has been immense over the years. Inherent with this growth came more and more HTTP servers. This is one of the reasons why the transition from RTP streaming with its problems to HTTP streaming from already deployed HTTP servers is so easy. However, this is not the only reason. HTTP streaming servers do not need to keep streaming session info on the server as the HTTP streaming client will be responsible for asking the right package on the right moment. The only thing the server has to do is respond with the package requested by the client. This is exactly what the HTTP server is made for. Because of these advantages of HTTP streaming, a lot of companies saw an opportunity to create their own implementation of the HTTP Adaptive Streaming (HAS) paradigm and make it the standard. Examples of these projects are Microsoft Smooth Streaming (MSS) [Mue15b], Apple HTTP Live Streaming (HLS) [Mue15a] and

Adobe HTTP Dynamic Streaming (HDS) [Ado]. As a result of every company developing their own implementation, there was a mixture of protocols that could not work together. In 2010 Dynamic Adaptive Streaming over HTTP (MPEG-DASH) was developed by the Moving Picture Experts Group (MPEG). In 2011 it became the international standard for streaming adaptive bitrate high quality media content over the Internet as defined by ISO/IEC 23009-1 [ISO14b].

3.2 HTTP MPEG-DASH streaming

As MPEG-DASH works on top of the existing network infrastructure, the HTTP streaming setup structure is simple as can be seen in Figure 3.1, where the flow from camera to client devices is shown in a typical situation [Sto11].

Once the media is recorded and ready to be streamed, the media processing and distribution on the Content Delivery Network (CDN) has to start. Before distribution is possible, the media file has to be segmented and the media info has to be written down in a MPD file. In this context *segmentation* means the process of dividing a media stream (e.g. video, audio, subtitles, ...) in smaller pieces of a fixed duration. Once the processing is done, the distribution on the Internet has to start by placing the Media Presentation Description (MPD) file with its corresponding media segments on a CDN.

A more concrete explanation of the complete syntax of a MPD file is given in Section 3.3. For now it is important to know that a MPD contains an Uniform Resource Locator (URL) for every segment that contains video or audio content. The URL will be different for every representation of the media content. This means that when the MPD file has been received and parsed, the DASH client can request the segments needed to play the media content over HTTP. In case of low bandwidth availability at client or server side, the video buffering time can be high. This can be resolved by changing quality and requesting lower quality segments to make sure the buffer always contains enough media content for smooth media consumption [Sod11]; nobody likes buffering and waiting for media content [DSA+11].

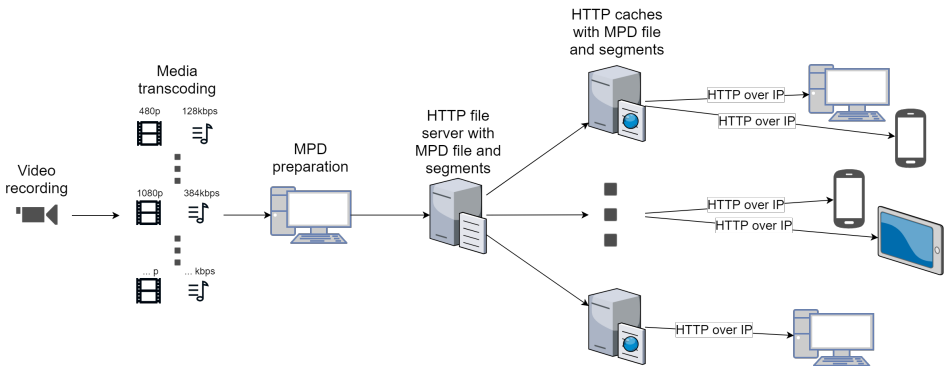


Figure 3.1: Simple MPEG-DASH flow

3.3 Media Presentation Description

When the MPD file is received on the client device, it has to be parsed. Within the parsing phase, every field of the MPD file gets extracted and interpreted to start the streaming session. In this master thesis the streamed video content is tile-based video. Because of the tiled nature of the content, the MPD file also includes Spatial Relationship Description (SRD) info. The SRD will be fully explained in Section 3.4. For now it is important to understand that SRD is extra info that does not need to be included in an MPD file. Afterwards, when the parsing phase is done, the media content streaming can start. Within this Section there will be an explanation of the structure of the MPD file and important fields.

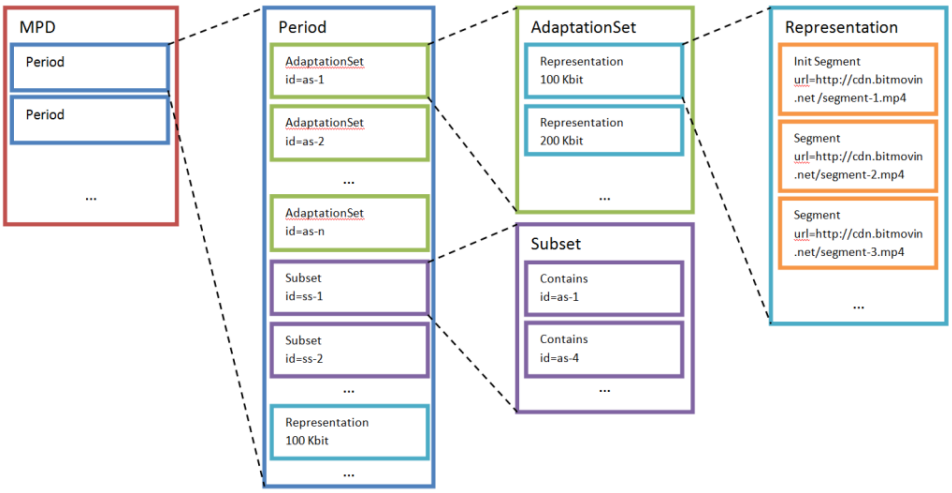


Figure 3.2: Media Presentation Description (MPD) model [Mue15c]

A MPD file has a specific structure, regardless of live or static content as can be seen in Figure 3.2. The file consists of a XML tree structure. At the top of this structure are **Periods**. Every **Period** is segment that is a representation of the media content of a specific non overlapping time interval. Contextually a **Period** contains codec parameters, server location or available variations of the content. As a result, a **Period** allows for splicing of media content to introduce ads or logical content segments. The *duration* attribute of the **Period** tag defines the start time relative to the start of the Media Presentation [Sto11]. An example of this logical decoupling between media segments within a sigle media presentation by means of the **Period** tag is this of an advertisement that is only available in high quality while the other media content have multiple bitrate representations. The advertisement will be described by a single **Period** containing only one **Representation**, while the media content in the periods before and after the ad contain multiple **Representations** [Mue15c].

A **Period** consists of one or more **AdaptationSets**. An **AdaptationSet** is a high-level representation of the media content. Each **AdaptationSet** represents a media stream. In a typical scenario, there exists one **AdaptationSet** for video and oen or more for audio. That way the audio can have an **AdaptationSet** for every language available. An **AdaptationSet** does not only support video and audio streams but also subtitles or metadata [Lon15].

Subsets are defined on the same XML depth level as **AdaptationSets** and are used

to specify which **AdaptationSets** can be combined. This gives the MPD designer the flexibility to limit combinations of **AdaptationSets**. Typically, a video and audio fragment of the same quality will be combined. For example, in some situations it is not satisfying to combine low resolution audio with high quality video (or vice versa) while streaming. By defining a **Subset**, the client is restricted in its **AdaptationSet** combination freedom. This is mostly needed for automated adaptation at client-side, based on the amount of bandwidth available. Consider the scenario where when a client has just enough bandwidth for really good audio but bad video; depending on the content, a user would in this case rather have medium quality audio and video. These kind of content combination restrictions can be enforced with **Subset**.

The next level into **AdaptationSet** are **Representations**. A **Representation** is a representation of media content in a specific quality. More precise, multiple **Representations** grouped in a single **AdaptationSet** for quality versions of the same content.

Every **Representation** is split up in one or more **Segments** with each their own URL. Each **Segment** contains a short segment of media content. By having many short **Segments**, a client can switch between content qualities in a smooth way without noticing a stutter or video reload. The reason for this is the interchangeability of **Representations** during the streaming session. For example, a **Segment** of a 100Kbits per second **Representation** can be used at startup of the stream. When the client notices that there is more bandwidth available, a second **Segment** can be requested from another **Representation** with for example a quality of 200Kbits per second. Even though the segments are from another **Representation**, they visualize the same content. It is also possible to have just one **Segment** describing the full content and is defined as a **SegmentBase**. However, this is a rare situation that will not be further explained. The typical duration of **Segments** are 2 and 10 seconds.

There are a few ways to represent **Segments** in the MPD file, one of those which has already been explained is **SegmentBase**. However, the two most used approaches are **SegmentList** and **SegmentTemplate**.

A more syntactic heavy approach is the **SegmentList**. Within this approach the URL is explicitly enumerated for every **Segment**. Every **Segment** has to be played in the same order as they are written in the MPD file. Listing 3.1 is an example of such a **Segment** declaration approach. As can be seen, this is verbose for media content with a substantial amount of **Representations** and **Segments**.

```
1  <Representation frameRate="25"
2                  bandwidth="1000000"
3                  width="1280" height="720">
4    <SegmentList timescale="1000" duration="2000">
5      <Initialization sourceURL="../video/720_1000000/dash/init.mp4"/>
6      <SegmentURL media="../video/720_1000000/dash/segment_0.m4s"/>
7      <SegmentURL media="../video/720_1000000/dash/segment_1.m4s"/>
8      ... 17 more entries
9      <SegmentURL media="../video/720_1000000/dash/segment_19.m4s"/>
10   </SegmentList>
11 </Representation>
12
13 <Representation frameRate="25"
14                 bandwidth="2000000"
15                 width="1280" height="720">
16   <SegmentList timescale="1000" duration="2000">
17     <Initialization sourceURL="../video/720_2000000/dash/init.mp4"/>
```

```
18      <SegmentURL media="../video/720_2000000/dash/segment_0.m4s"/>
19      <SegmentURL media="../video/720_2000000/dash/segment_1.m4s"/>
20      ... 17 more entries
21      <SegmentURL media="../video/720_2000000/dash/segment_19.m4s"/>
22    </SegmentList>
23  </Representation>
```

Listing 3.1: A simple **SegmentList Representation** example

SegmentTemplate is a more compact approach. An example can be seen in Listing 3.2. The *media* attribute of the **SegmentTemplate** will define an URL, the template URL. The template URL will have a variable that is modified based on the needed representation. In the example, the `id` of one of the two **Representations** will be taken and used to fill in the *RepresentationID* parameter of the media URL. Similarly, the desired segment index will be used to substitute the number template parameter.

```
1
2   <SegmentTemplate media="../video/$RepresentationID$/dash/
        segment_$Number$.m4s"
3                     initialization="../video/$RepresentationID$/dash/
                          init.mp4"
4                     timescale="1000" duration="2000"
5                     startNumber="0"/>
6
7   <Representation id="720_1000000" frameRate="25" bandwidth="1000000"
8                   width="1280" height="720"/>
9
10  <Representation id="720_2000000" frameRate="25" bandwidth="2000000"
11                  width="1280" height="720"/>
```

Listing 3.2: A simple **SegmentTemplate Representation** example

3.3.1 Comprehensive example

Listing 3.3 is an example of a simple MPD file. The easiest way to scan a MPD file manually for the content of the stream is to search for the **Periods**. As can be seen, this file contains only one **Period** with 2 **AdaptationSets**. A quick look also reveals the use of **SegmentTemplate** and not the syntax heavy form, **SegmentList** to describe media segment URLs.

When looking at the attributes of the MPD tag, notice that this MPD file is *static* as defined by the attribute `type`. This means that the MPD file does not need to be updated as would be the case in a *dynamic* `type` (e.g., live content). Another attribute of the MPD tag is `mediaPresentationDuration` (see the third line of the listing). With this attribute the duration of the complete static MPD file is defined. The duration is defined in xsd:duration as can be found in ISO/IEC 23009 [ISO14b]. In this example, the duration of the described media presentation is 46 seconds. Lastly, there is the `minBufferTime` attribute. As the name states, this is the minimum buffer time suggested by the MPD which is 1 second in this example. The `minBufferTime` is, same as `mediaPresentationDuration`, specified in the ISO/IEC 23009 [ISO14b] time definition.

As can be seen in Listing 3.3 and explained in Section 3.3, the next tag will be **Period**. Within the **Period** tag, two **AdaptationSets** are defined. The one defined on line

6 is a video **AdaptationSet**, while the one on line 17 is an audio **AdaptationSet**. As mentioned before, both **AdaptationSets** are defined with a **SegmentTemplate** definition.

Considering the first **AdaptationSet** on line 7, there are 2 possible **Representations** in the same video resolution. With a bandwidth of 3 000 000 and 1 000 000 bits per second, respectively. While the streaming session is in play, the client can choose to switch between these two **Representations** depending on the available bandwidth or client settings. The attribute `timescale` of the tag **SegmentTemplate** represents the amount of ticks per second. Together with attribute `duration` the duration of one **Segment** (in seconds) can be calculated with the following equation:

$$segment\ duration = \frac{\texttt{duration}}{\texttt{timescale}}$$

This gives a duration of 2 seconds for every **Segment** in this example.

For the second **AdaptationSet** there is no real difference besides the fact that this is an audio **AdaptationSet** for English content and only contains a single **Representation** of 128 000 bits per second.

Dit weg doen en mss gaan naar een voorbeeld met SRD informatie erachter maar gaat mss al te complex zijn omdat dan ook tiling aanbod komt.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <MPD id="mpd-file" type="static"
3      mediaPresentationDuration="P0Y0M0DT0H0M46.000S"
4      minBufferTime="P0Y0M0DT0H0M1.000S">
5      <Period>
6        <AdaptationSet mimeType="video/mp4" codecs="avc1.42c00d">
7          <SegmentTemplate
8            media="../video/$RepresentationID$/dash/segment_$Number$.m4s"
9            initialization="../video/$RepresentationID$/dash/init.mp4"
10           duration="60000" timescale="30000" startNumber="0" />
11         <Representation id="800_3000000" bandwidth="3000000"
12           width="2400" height="800" frameRate="30"/>
13         <Representation id="800_1000000" bandwidth="1000000"
14           width="2400" height="800" frameRate="30"/>
15       </AdaptationSet>
16
17       <AdaptationSet mimeType="audio/mp4" codecs="mp4a.40.5" lang="en">
18         <SegmentTemplate
19           media="../audio/$RepresentationID$/dash/segment_$Number$.m4s"
20           initialization="../audio/$RepresentationID$/dash/init.mp4"
21           duration="60000" timescale="30000" startNumber="0" />
22         <Representation id="128000" bandwidth="128000"/>
23       </AdaptationSet>
24     </Period>
25 </MPD>
```

Listing 3.3: Comprehensive example of a MPD-file

3.4 Spatial Relationship Description

Throughout this chapter, the structure of a basic MPD has been explained. However, to allow for tiled streaming (which is fully explained in Chapter 4) an extension is needed upon the basic MPD structure. This new extension came with the second amendment of MPD in ISO/IEC 23009-1:2014/Amd.2:2015 to identify Spatial Relationship Description (SRD) [iso15].

The need to define a spatial relationship in the MPD file grew as Region of Interest video streaming became more popular. Chapter 4 gives more detail about Region of Interest streaming, more specifically about tiled video. By providing SRD information, the DASH-client knows what video tiles to request based on the client's viewport location in the video. Based on this location, a DASH client can choose to stream all or only a subpart of the video tiles provided. This allows users to stream a subset of the video in high quality or by zooming in or out to view the complete video in a lower quality [DvdBTN16].

The SRD amendment adds a few new features to the MPD syntax. A complete description can be found in ISO/IEC 23009-1 [iso15]. Only the important subjects in context of this master thesis will be explained. The main concept is to define a 2 dimensional space for the different locations of the media objects.

Two new tags have been introduced by the SRD: the **EssentialProperty** and **SupplementalProperty**. Furthermore, both properties are defined in an **AdaptationSet**. The **EssentialProperty** allows the MPD author to define that it is essential to successfully processing this descriptor to ensure a correct processing of the parent content, which is a **AdaptationSet**. When legacy DASH clients encounter this descriptor, they will discard the video content [NTD+16]. With the **SupplementalProperty**, the MPD author can define that it is not essential to correctly parse the information of the descriptor to ensure correct processing the the parent content.

These tags define two attributes: *schemeIdUri* and the *value*. The *schemeIdUri* defines the scheme when reading values. In this context the value of *schemeIdUri* will be `urn:mpeg:dash:srd:2014`. The *value* tag is a bit more complexly structured and contains the following fields separated by comma [NTD+16]:

- *source_id* is a required field that exists out of an integer value identifying the contents source. With this identifier a tile can be referenced to a video.
- *object_x* a non-negative integer and required field representing the horizontal position starting from the top-left corner of the corresponding media asset.
- *object_y* a non-negative integer and required field representing the vertical position starting from the top-left corner of the corresponding media asset.
- *object_width* is a required field which represents the width of the corresponding media asset with a non-negative integer.
- *object_height* is a required field which represents the height of the corresponding media asset with a non-negative integer.
- *total_width* is an optional field representing a non-negative integer which expresses the width of all media assets of the same identifier.

- *total_height* is an optional field representing a non-negative integer which expresses the height of all media assets of the same identifier.
- *spatial_set_id* is the last optional non-negative integer which provides an identifier for a group of media assets.

3.4.1 Comprehensive SRD example

A simplified version of a MPD with SRD information can be seen in Listing 3.4. Note that this example resembles a real use case and has been used in Chapter 6 to test the set up. In listing there is a definition of a 3 by 3 tiled video with a 4K resolution.

As the SRD is built upon MPD and is created as an extension, it was of utmost importance that clients with no knowledge of SRD info could still work with the MPD file, keeping backwards compatibility. Therefore, the extension of two attributes. Because regular MPD tags have already been explained and used in the comprehensive example of Section 3.3.1, no further explanation about these tags will be given and the focus will be on the SRD information.

Under every **AdaptationSet** there is either the **EssentialProperty** or **SupplementalProperty**. These tags, as described in Section 3.4, are defined by the ISO/IEC 23009-1 amendment 2 [iso15]. Structurally these tags contain two attributes *schemeIdUri* and the *value*. In the example provided, the value of the *schemeIdUri* attribute is always *urn:mpeg:dash:srd:2014*.

The **AdaptationSet** on line 4 holds an **EssentialProperty** (on line 5), which has as *value* attribute `1,0,0,0,0`. The *source_id* is 1 and is the same for all properties (see line 9, 13, 17, 21). The other values of the *value* attribute consist of the x-, y-location, width and height. The reason for defining the width and height 0 in this **AdaptationSet** is the content that it represents. This **AdaptationSet** consists of only header info which will be used to recreate the complete video frame, a more detailed explanation is given in Chapter 5. For now it is good enough to understand that this **AdaptationSet** only contains header info needed to decode the video.

The next SRD property to be found is the one in the second **AdaptationSet** on line 8. This **AdaptationSet** contains a **SupplementalProperty** with the values *1,0,0,1280,704*, defining that the x- and y-locations are 0, the width is 1280 pixels and height 704 pixels. The other **SupplementalProperty** of the other **AdaptationSets** will have similar values based on their 2 dimensional location. A reasoning behind the width and height of the tiles and why the MPD has 10 **AdaptationSets** for 9 tiles is given in Chapter 5.

In our case, there is no *total_width* or *total_height* defined. This is because there is only one reference space, extending the whole video.

```
1 <?xml version="1.0"?>
2 <MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.000S"
     type="static" mediaPresentationDuration="PT0H0M22.583S"
     maxSegmentDuration="PT0H0M1.000S" profiles="
     urn:mpeg:dash:profile:full:2011">
3  <Period duration="PT0H0M22.583S">
4   <AdaptationSet segmentAlignment="true" bitstreamSwitching="true"
       maxWidth="3840" maxHeight="2160" maxFrameRate="24000/1001" par="
       16:9" lang="und">
```

```
5     <EssentialProperty schemeIdUri="urn:mpeg:dash:srd:2014" value="
          1,0,0,0,0"/>
6     ... Representation info
7    </AdaptationSet>
8    <AdaptationSet ... >
9     <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014" value="
          1,0,0,1280,704"/>
10   ... Representation info
11   </AdaptationSet>
12   <AdaptationSet ... >
13   <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014" value="
         1,1280,0,1280,704"/>
14   ... Representation info
15   </AdaptationSet>
16   <AdaptationSet ... >
17   <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014" value="
         1,2560,0,1280,704"/>
18   ... Representation info
19   </AdaptationSet>
20   <AdaptationSet ... >
21     <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014" value="
           1,0,704,1280,704"/>
22   ... Representation info
23   </AdaptationSet>
24   ... 5 more AdaptationSet with their own values
25  </Period>
26 </MPD>
```

Listing 3.4: Comprehensive example of a MPD-file with SRD extensions

3.5 DASH-Client

As previously mentioned in Section 3.2 is the DASH-client an essential part of MPEG-DASH streaming. The DASH-client will handle all client side actions which represents the web browser in Section 5.3. Therefor, will the explanation in this section be in the use case of this master thesis.

The client will ask for the MPD file to start the streaming session, this can be via any medium (e.g. Internet). In the context of this master thesis, this will be with a HTTP GET request to the server containing the MPD file. Once the MPD file has been received, the parsing and extraction of information begins. The SRD information will be disregarded to simplify the explanation.

After the information extraction, a `scheduler` is initialized. The `scheduler` functionality consists of requesting the correct segment at the correct time. There are a few ways to implement a `scheduler`, one of those is the *steady fill* approach. Within this approach, the client looks at the buffered time, current play time and the minimum buffer time described in the MPD. Once the the sum of the current playtime and the minimum buffer time is higher than the buffered time, the scheduler will make a request for the new segment(s). The quality of the requested segments is depending on the viewport location corresponding to the tiles. A more detailed explanation of the quality selection process is given in Section 5.3. When these segments are received, they will be added to the video buffer to be played.

Chapter 4

TILED Video

Tiled video has been the subject of a lot of research in the field of omnidirectional (360 degree) video and Virtual Reality (VR). Major platforms like Facebook and YouTube have also started streaming 360 degree video to a wide variety of devices. However, these omnidirectional videos require tremendous amounts of bandwidth [SSP+17]. The high bandwidth is linked to the multiple times high quality content that is needed. The resolution can easily go up to 12,288 x 6,144 pixels for 4K content. This high bandwidth needed is in contradiction with the fact that the user only sees a part of the video, from 96 degree to 110 degree of the omnidirectional video for head mounted devices [OAS17]. Some other situations allow the viewport to be increased or decreased which is the case in our implementation, further discussed in Chapter 5. However, the complete video is needed for when the user turns around with a head mounted display and changes his region of interest. The region of the video that is being viewed is called the viewport.

The most common technology used to be able to provide data intensive 360 degree video over the Internet is Dynamic Adaptive Streaming over HTTP (MPEG-DASH). With MPEG-DASH a user is able to adaptively stream different qualities based on the available bandwidth. A more detailed explanation of MPEG-DASH is given in Chapter 3. In some cases, other manners of transporting the media content is used than MPEG-DASH. However, in the related work review (Section 4.1) we only discuss situations where also MPEG-DASH is used as this thesis conducts work based on MPEG-DASH.

4.1 Related work

The main idea within tiled 360 degree video streaming is always the same: try to limit the high quality content to the viewport. A simple solution to making sure no extra content is streamed is by sending only the video that can be seen in the viewport. However, what if a user moves the viewport to another location? An unwanted situation would then be a blank screen for the maximum of one segment duration. This is why a lot of research is begin done in the domain of tiled streaming and also the reason why tiles that are not in the viewport are transmitted in a lower quality.

4.1.1 Equirectangular encoding

In approaches of tiled streaming where the tile support was not natively included in the video codec, multiple decoders were needed. These decoders would each decode a tile. Once all tiles were decoded, a synchronization step between the different video elements were needed to ensure a synchronized playback. In [MAP+10] such implementation is presented in an online lecture context.

Allowing only a subset of the video to be streamed in high quality, more precisely the viewport, the bandwidth usage can be reduced. By adaptively giving a quality based on the current user's viewport and less to the surrounding video, can give a promising visual quality increase [SSHS16].

At one of the last stages of the thesis the [GTM17] paper was found. They also implemented a web browser implementation of the MPEG-DASH tiled HEVC stream to a single bit stream. However, no information is given on the overhead provided by the remuxing needed to playback the HEVC stream nor how the web application works. They did however measure the impact of bandwidth management and the amount of tiles which they found best for bandwidth versus quality in omnidirectional context for different devices. In contrast, this thesis will give an overview how a web application can be made from open source project *FFmpeg* and analyze the overhead to reproduce a single HEVC bit stream that is decodable with one hardware decoder to be able to stream MPEG-DASH tiled HEVC video in the web browser.

Another paper that was found in one of the later stages of the thesis was [CFD+17]. In this paper an open source implementation is provided. Furthermore, in this paper the bandwidth streaming size overhead is measured versus non tiled video. This paper is produced by the same people that made the guide on how to create the HEVC tiled MPEG-DASH content which this thesis is based on (see Section 5). However, this thesis is unique in regards of the web browser implementation based on the open source project *FFmpeg*. Also, within this thesis the overhead to preprocess a tiled MPEG-DASH stream to a natively tiled video stream is measured in regards to resolution and amount of tiles as can be seen in Chapter 6.

4.1.2 Cubical encoding

In [SSP+17], which is the follow-up paper of [SSHS16], they presented an approach were the resolution outside the viewport is decreased on-the-fly. They do this by using motion-constrained HEVC tiles. These tiles are streamed using MPEG-DASH in which they allow for adaptive resolution of the individual video areas based on the user viewport. Tiles outside the viewport will be in lower quality. The tiles of varying quality are eventually merged into a single bit stream. In contrast with this thesis where equirectangular projection is used, will the tiles be cubical formed. A visual representation can be seen in Figure 4.1 They believe that a cubic approach fits better into the field of view of head mounted devices. At server side they present segments in two variations, one segment is a segment with the random access point which is needed when a tile quality is changed. To counter bitrate peaks, a second segment with the exact same content is provided without the random access point. These papers lacked information on how the content creation was done. Also, was no further information given on how the merging of tiles was done besides lightweight tile aggregation as described in [SSS15].

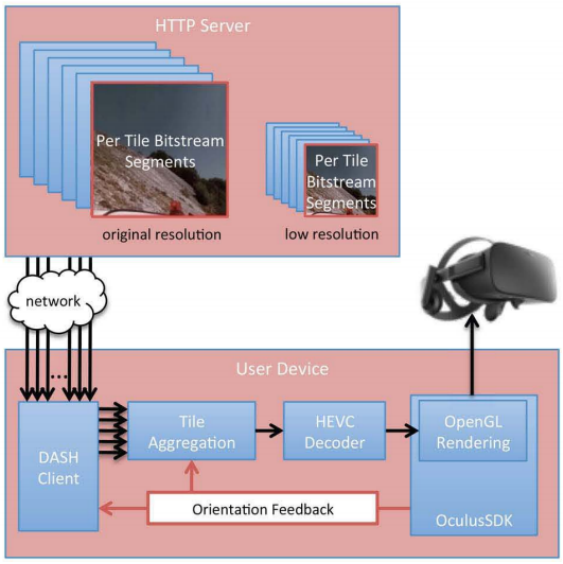


Figure 4.1: Cubical devision of the tiles in a demonstrator overview [SSP+17]

Another approach is given in [HS16] and the follow up [HS17]. In this paper they introduce a hexaface sphere which is a underlying 3D geometry mesh of the omnidirectional video. They do this in two parts. Firstly they partition the video in tiles and use the SRD extension of MPEG-DASH. Secondly, they cut the 3D mesh in which the video is projected into 3 major parts being the top, middle and bottom. After this, they cut the middle part in 4 pieces of equal size which is a 90 degree view which can be seen in Figure 4.2. To map the tiles on the created hexaface sphere, they use a mapping mechanism to map the 6 tiles (one for every part of the sphere) on the surfaces. By using viewport tracking, they dynamically deliver high bitrate content to the tiles in the viewport and lower qualities in the surrounding tiles.

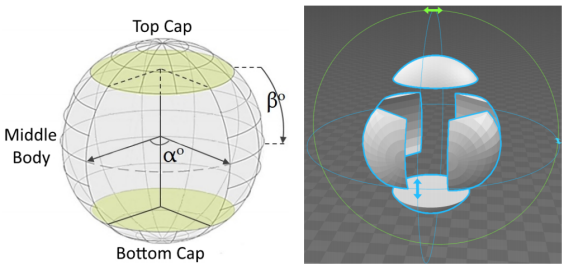


Figure 4.2: Visual representation of the generated hexaface sphere [HS17]

4.1.3 Pyramid encoding

An approach that was taken by Facebook is viewport oriented encoding based on the user's head position with a head mounted device [Kuz16]. They do this by using pyramid encoding and cube faces instead of equirectangular layouts. A pyramid represents the viewport of the user. The pyramid is placed within the 360 degree viewing orb. At the base of the pyramid is the high quality image within the viewport. The sides of the pyramid are unwrapped and the cube is stretched to fit the whole 360 degree view of a user. The sides of the cube have a degrading quality going from the bottom of the

pyramid to the top. However, for every direction in which the user can look is a new pyramid needed so these pyramids have to be generated. They implemented 30 pyramids to cover the whole 360 degree viewing sphere. When a user moves his/her head, he/she will be placed in a new pyramid. This is done by different representations of the same video for every pyramid. For example, 5 different representations of the video with 30 pyramids needs 150 representations of the same video content. This comes with major overhead with respect to the generation, encoding and storage of the same original content to allow the DASH client to switch between representations.

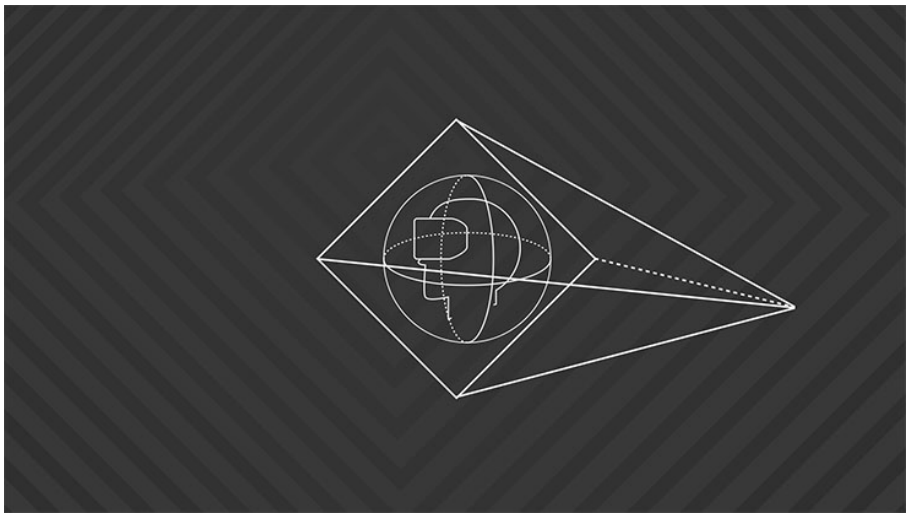


Figure 4.3: Facebook's approach for viewport oriented encoding based on the user's head position with pyramids [Kuz16]

Chapter 5

Implementation

While the tiled subject in context of adaptive video with MPEG-DASH has been researched a lot in H.264/AVC, especially in the world of Virtual Reality (VR) and 360 degree video as mentioned in Chapter 4, there is yet to be a clear definition of tiled HEVC in MPEG-DASH. Therefore, the implementation has been a path of many trials and errors.

The implementation started with a simple idea; *Stream tiled HEVC video over MPEG-DASH to project 360 degree video so one decoder is needed and adaptive control of quality is possible.* This is in contrast with H.264/AVC, were the need for every tile to have its own decoder and a tile manager to ensure a synchronized playback between the processed tiles is needed. This subject is further discussed in Chapter 4 and Chapter 2 were a full explanation of the HEVC video codec is given and compared with H.264/AVC. By implementing this *simple* idea, a user would be able to adapt the quality of the video when looking at the 360 degree video based on the viewport of the user while fully utilizing the hardware acceleration for HEVC decoding.

It was know that *GPAC* had done some work around tiled HEVC adaptive MPEG-DASH video streaming and had already implemented and demoed such implementation at scientific conferences. However, the only thing that was found at the start of the thesis were contradicting tutorials on generating tiled HEVC MPEG-DASH content, [Feu17] and [Feu18]. The first tutorial was found on their website, the second tutorial was found on the *GitHub* page of the project. It was only after understanding the complete syntax of the bit stream and asking for an explanation by creating an issue on the *GitHub* of the project that the full generation process was clear. The process will be described in detail in Section 5.1 and will hopefully be used by the *GPAC* team to update their tutorial to a consistent guide for creating tiled HEVC MPEG-DASH content. Furthermore, was it only in the last step of the implementation, the *JavaScript* implementation described in Section 5.3, before the faults in the guide were clear. This is because the tests were primarily done by content in the same quality or content made available by the *GPAC* team. Another struggle of the implementation by the *GPAC* team was that is was a native application for desktop. The purpose of this master thesis implementation was to be an web application for use in web browser context.

Once the course was clear, the decision was made to start with a native application as the support for HEVC in web browser context was so minimal that only Edge was supporting HEVC on Windows and Safari on Mac OS. Apple announced the support of HEVC in

its yearly keynote of 2017 [Inc18]. However, the release would hold off until the new big Mac OS X update, more specific: update 10.13 High Sierra released September 25, 2017 to the public. With this update the support for HEVC in Safari. In both web browser there were restriction of not allowing tiled videos in separate streams.

Therefor, the choice to make a web application that could create from MPEG-DASH segments that contained the tiled HEVC bit stream to a tiled HEVC video with one tiled stream so the web browser could decode the video bit stream was made. However, creating such a complex web application is hard. Especially when working with a complex video codec as HEVC, which has been described in Chapter 2. Therefor, the choice to start with a native application was made.

In Section 5.2 the choices and implementation of the native application will be informed and discussed. After the native application, the conversion process of the native application to a working web browser implementation will be explained. A complete flow of the program as implemented in the final *JavaScript* product can be seen in Figure 5.7.

5.1 Content preparation

The content preparation is based on the guides from *GPAC* [Feu17] [Feu18]. However, these guides are outdated and mis key information to create tiled HEVC content streamable over MPEG-DASH that allows for mixing qualities. In subsection 5.1.1 a detailed example will be given that was used as an video to test the implementation in this master thesis.

The preparation of the content has been done with *Kvazaar* version 1.2.0 [kva18], *FFmpeg* version 2.8.14 [FFm18] and MP4Box version 0.7.1 [GPA18] which is a implementation by *GPAC*. Make sure the same versions are used to generate tiled HEVC MPEG-DASH content with this tutorial.

The content creation process starts by taking a video recording. This can be any kind of video in any format. We start by decoding the video to raw YUV-format. A more extensive explanation of the YUV color space can be found [Mic18]. For now, it is important to know what YUV means:

- Y stands for the luminance which is the amount of light, leaving from a point in any direction being that it is emitted, passes through or reflected from a surface.
- U stands for the horizontal axis in the color space representing the blue plane.
- V stands for the vertical axis in the color space representing the red plane.

The conversion of the video to a raw YUV video is because of *Kvazaar*. *Kvazaar* expects input in YUV420p format at 8 bit depth. Understanding that 4:2:0 stands for only half of the bits going to the color space while double the bits go to the luminance is enough for the YUV content. Furthermore, represents the 8 bit depth the amount of bits used to save the luminance and color.

To convert the video from any video in any container to a raw YUV420p 8 bit depth video, *FFmpeg* can be used. A command for this can be seen in Listing 5.1.

- *-i* defines the input video.

- *-c:v* is a bit more complicated as the *-c* stands for defining the codec and *:v* is a way of defining that this codec is specifically meant for video. The encoder that will be used is *rawvideo*.
- *-pix_fmt* defines the pixel format; as previously explained, this has to be YUV420p.
- *<output_video.yuv>* is the last parameter which is the name of the output video.

```
$ ffmpeg -i <input_video> -c:v rawvideo -pix_fmt yuv420p <
    output_video.yuv>
```

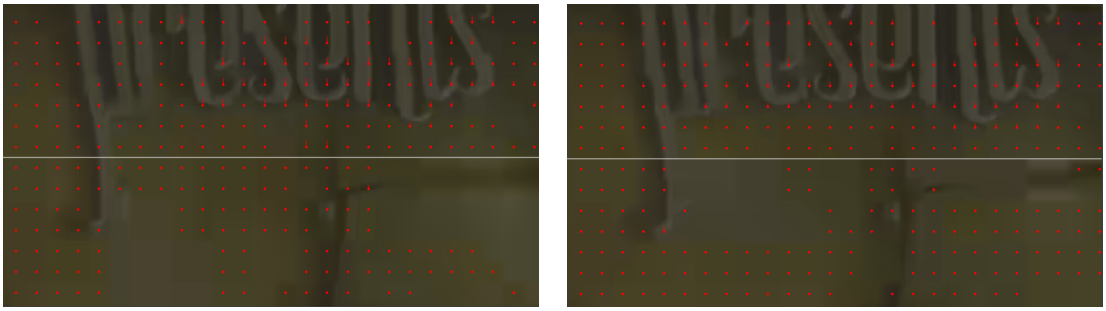
Listing 5.1: Command to convert a video in a container to a raw YUV420p video with FFmpeg

Once the raw video has been created, it can be used as input for *Kvazaar*. *Kvazaar* will take the raw video as input and create a HEVC tiled video. A more detailed explanation of HEVC and tiles can be found in Chapter 2. The command to go from a raw video in YUV420p 8bit depth format to a HEVC tiled video can be seen in Listing 5.2.

- *-i* the input file containing the raw YUV420p video.
- *--input-res* the resolution of the input video, in pixels.
- *-o* the output file, this will be a tiled HEVC video.
- *--tiles* the amount of tiles that has to be used in *Width*x*Height*. See Section 2.2.1 for a detailed explanation of the allowed amount of tiles in width and height.
- *--slices* A detailed explanation of how tiles can be encoded in HEVC can be found in Section 2.2. For now it is important to know that this parameter defines the way the slices have to be defined. The parameter used here is `tiles`, which stands for putting the tiles in independent slices.
- *--mv-constraint* With this parameter the user can define how the motion vectors should be restricted. There are two options, `frametile` and `frametilemargin`. `frametile` will restrict the motion vectors within the tiles. `frametilemargin` will be even more restrictive. A comparison between the two can be seen in Figure 5.1.
- *--bitrate* the bitrate per seconds abide the tiled HEVC video should be encoded in.
- *-q* the quantization parameter that has to be used. This parameter will be used to write in the VPS header (see Chapter 2 for a detailed explanation of the VPS and other HEVC headers) but will be disregarded in the encoding process if the `bitrate` parameter has been set. In Subsection 5.1.1 this will become more clear. Setting this parameter, even if it has no impact on the encoding process, is of paramount importance for the subsequent MPEG-DASH segmentation step to work properly (see later).
- *--period* the period of intra pictures, in frame count. This should be consistent with the MPEG-DASH settings later on.
- *--input-fps* the input framerate of the raw YUV420p video.

```
$ kvazaar -i <input_video.yuv> --input-res 3840x2160 -o <
    output_video.hvc> --tiles 3x3 --slices tiles --mv-constraint
    frametilemargin -q 30 --bitrate <bitrate> --period 30 --input-fps 30
```

Listing 5.2: Convert a raw YUV420p input video to a HEVC tiled video.



(a) A zoomed in still where the setting *frametile* has been used. Notice how the motion vectors are bigger than in the *frametilemargin* setting

(b) A zoomed in still where the setting *frametilemargin* has been used. Notice how the motion vectors are smaller than in the *frametile* setting

Figure 5.1: Two stills showing the exact same frame with the *frametile* and *frametilemargin* options for the *--mv-constraint* parameter of *Kvazaar*.

Next step is using MP4Box, provided by the team of GPAC themselves. Even though this toolkit is one with many more features [GPA18], MP4Box will only be used to create a MP4 file which has all tiles split up in different tracks from the created HEVC in the previous step. In Listing 5.3 the command to create a MP4 is given. With this command the *input_video.hvc* will be split up by means of tiles. Each tile will be assigned to a MP4 stream. However, there will be a track 1 that contains all non-VCL NAL units of the created MP4. This means that the generated MP4 file will have *amount of tiles + 1*-streams. This extra track will contain all non VCL NAL units needed to process the tiles. A detailed explanation of the NAL units is given in Section 2.5 and a complete overview of the HEVC bit stream is given in Chapter 2. For now it is good enough to understand that each track contains one tile region, besides track 1 which contains only non-VCL NAL units giving a total of *amount of tiles* + 1 tracks to the generated MP4. When using this command it is import to make sure the FPS values are the same as in other commands.

```
$ MP4Box -add <input_video.hvc>:split_tiles -fps 30 -new <
    output_video.mp4>
```

Listing 5.3: Convert a HEVC tiled video into a MP4 which defines a stream for every tile +1 extra stream for NAL units

Last step in the generation of tiled HEVC MPEG-DASH content is the creation of the MPEG-DASH content from the created HEVC tiled MP4 with MP4Box. This process is illustrated in Listing 5.4.

- *-dash* defines the length of the segments in milliseconds.
- *-rap* with this parameter, the user defines that the program has to start every segment with an I-frame so it is possible to start decoding and replaying the video form each received segment.
- *-segment-name* represents the naming format. By including %s_segment, the user defines that the name of the segments should start with the original input filename (%s) and then be concatenated with the fixed string '_segment'. Once the name formatting is done, the name is appended with a $number$ which will represent the number of the segment, starting from 0 till the last segment in increments of 1.

- *-min-buffer* specifies in the generated MPD how long the minimum buffer time should be.
- *-url-template* defines that the segments in the MPD should be specified using the a `SegmentTemplate` syntax. More info about the MPD syntax can be found in Chapter 3
- *-out* the name of the generated MPD file.

```
$ MP4Box -dash 5000 -rap -segment-name %s_segment -min-buffer 2000 -url-
    template -out <output_mpd_file.mpd> <input_video_quality_1.mp4> <
    input_video_quality_2.mp4> ... <input_video_quality_N.mp4>
```

Listing 5.4: Generation of a tiled MPEG-DASH streamable video on basis of tiled MP4 videos which contain a stream per tile + 1 NAL unit stream

5.1.1 Comprehensive example

The first step is the decoding a video of to raw YUV-format or starting with a raw YUV-format. In this example, the YUV-format is already acquired. As the YUV file is already acquired and is named *elephants_dream.yuv*, the encoding to tiled HEVC can start with *Kvazaar* as seen in Listing 5.5.

```
$ kvazaar -i elephants_dream.yuv --input-res 3840x2160 -o 3840
    x2160_fps24_frametile_600000.hvc --tiles 3x3 --slices tiles --mv-
    constraint frametile -q 30 --bitrate 600000 --period 24 --input-fps
    24
```

Listing 5.5: Conversion example of a raw YUV input video to a HEVC tiled video

The output of the previous command can be seen in Figure 5.2. Notice how the Quantization Parameter fluctuates and is nowhere near the set value of 30. This is as previously mentioned because of the *bitrate* value being set. As *Kvazaar* tries to maintain the encoding bit rate specified, it is impossible to hold a steady QP value of 30. However, the QP value of 30 will be set in the PPS NAL unit of the HEVC stream as mentioned in Chapter 2. The details of the different NAL units are given in Section 2.5. Setting the PPS NAL unit to 30 is needed because of the tiles being split up in their own streams in the next step. When the tiles are split up, one stream with the non VCL NAL units is generated. This stream has to be the same in the different generated MP4 files for the next steps to work. If the VPS, SPS and PPS header would not be the same, the splitting in different streams and having one stream with the parameter sets (NAL units), would not be possible.

Next step is the making of the MP4 container with a stream per tile and one stream for the non VCL NAL units. This is done in Listing 5.6. The output video in the MP4 container can be seen in Figure 5.3. Notice how *Video stream 1* contains the complete video information as if no tiles were present. When zoomed in on any of the other video streams, there is the information per tile. Something important to notice here is that the video resolution, more precise the height, changes for the last row. This means that the first 3 tiles will have a resolution of 1280x704 pixels, the second row is 1280x704 pixels and the third row is 1280x752 pixels. *Kvazaar* tried to divide the video in blocks of 64 pixels but 2160 is not divisible by 64. This is why the last tile row has a bigger size being, 752 pixels. This is fixed by giving one CTB (see Chapter 2 for more info) 48 pixels in height instead of 64.

```
Input: 3840x2160_fps24.yuv, output: output_video.hvc
  Video size: 3840x2160 (input=3840x2160)
POC    0 QP 48 (I-frame)      71120 bits PSNR Y 30.8682 U 36.3148 V 34.1412
POC    1 QP 51 (P-frame)      12984 bits PSNR Y 30.9355 U 36.3048 V 34.1329 [L0 0 ] [L1 ]
POC    2 QP 51 (P-frame)       7048 bits PSNR Y 30.9465 U 36.3043 V 34.1319 [L0 1 ] [L1 ]
POC    3 QP 51 (P-frame)       9928 bits PSNR Y 30.9606 U 36.3023 V 34.1269 [L0 2 ] [L1 ]
POC    4 QP 48 (P-frame)       9592 bits PSNR Y 30.9838 U 36.2951 V 34.1222 [L0 3 ] [L1 ]
POC    5 QP 51 (P-frame)       6576 bits PSNR Y 30.9844 U 36.2934 V 34.1217 [L0 4 ] [L1 ]
POC    6 QP 51 (P-frame)       6480 bits PSNR Y 30.9845 U 36.2953 V 34.1226 [L0 5 ] [L1 ]
POC    7 QP 50 (P-frame)       6936 bits PSNR Y 30.9888 U 36.2932 V 34.1203 [L0 6 ] [L1 ]
POC    8 QP 47 (P-frame)       7560 bits PSNR Y 31.0017 U 36.2969 V 34.1266 [L0 7 ] [L1 ]
POC    9 QP 51 (P-frame)       6496 bits PSNR Y 31.0015 U 36.2961 V 34.1284 [L0 8 ] [L1 ]
POC   10 QP 47 (P-frame)       7488 bits PSNR Y 31.0133 U 36.2970 V 34.1276 [L0 9 ] [L1 ]
POC   11 QP 49 (P-frame)       7120 bits PSNR Y 31.0192 U 36.2974 V 34.1373 [L0 10 ] [L1 ]
POC   12 QP 50 (P-frame)       6528 bits PSNR Y 31.0171 U 36.2974 V 34.1373 [L0 11 ] [L1 ]
POC   13 QP 45 (P-frame)      21520 bits PSNR Y 31.1945 U 36.3103 V 34.1752 [L0 12 ] [L1 ]
POC   14 QP 46 (P-frame)       7608 bits PSNR Y 31.2082 U 36.3116 V 34.1780 [L0 13 ] [L1 ]
POC   15 QP 51 (P-frame)       6504 bits PSNR Y 31.2069 U 36.3104 V 34.1767 [L0 14 ] [L1 ]
POC   16 QP 42 (P-frame)      75880 bits PSNR Y 32.1628 U 36.8815 V 34.7564 [L0 15 ] [L1 ]
POC   17 QP 44 (P-frame)       8968 bits PSNR Y 32.1743 U 36.8782 V 34.7520 [L0 16 ] [L1 ]
POC   18 QP 51 (P-frame)       6536 bits PSNR Y 32.1726 U 36.8798 V 34.7526 [L0 17 ] [L1 ]
POC   19 QP 51 (P-frame)       6504 bits PSNR Y 32.1703 U 36.8791 V 34.7527 [L0 18 ] [L1 ]
POC   20 QP 37 (P-frame)     298136 bits PSNR Y 34.7478 U 40.4361 V 38.8262 [L0 19 ] [L1 ]
POC   21 QP 51 (P-frame)       6520 bits PSNR Y 34.7403 U 40.4355 V 38.8232 [L0 20 ] [L1 ]
POC   22 QP 51 (P-frame)       6536 bits PSNR Y 34.7388 U 40.4343 V 38.8246 [L0 21 ] [L1 ]
POC   23 QP 51 (P-frame)       6632 bits PSNR Y 34.7354 U 40.4342 V 38.8222 [L0 22 ] [L1 ]
POC   24 QP 46 (P-frame)       6720 bits PSNR Y 34.7316 U 40.4332 V 38.8186 [L0 23 ] [L1 ]
POC   25 QP 51 (P-frame)       6520 bits PSNR Y 34.7259 U 40.4328 V 38.8189 [L0 24 ] [L1 ]
POC   26 QP 48 (P-frame)       6592 bits PSNR Y 34.7243 U 40.4305 V 38.8174 [L0 25 ] [L1 ]
POC   27 QP 46 (P-frame)       6736 bits PSNR Y 34.7244 U 40.4345 V 38.8209 [L0 26 ] [L1 ]
POC   28 QP 39 (P-frame)      26744 bits PSNR Y 34.7674 U 40.4379 V 38.8118 [L0 27 ] [L1 ]
```

Figure 5.2: Output when using the commands of Listing 5.5

```
$ MP4Box -add 3840x2160_fps24_frametile_600000.hvc:split_tiles -fps 24 -
    new 3840x2160_fps24_frametile_300000.mp4
```

Listing 5.6: Create a MP4 container with one stream per tile and a non VCL NAL unit stream

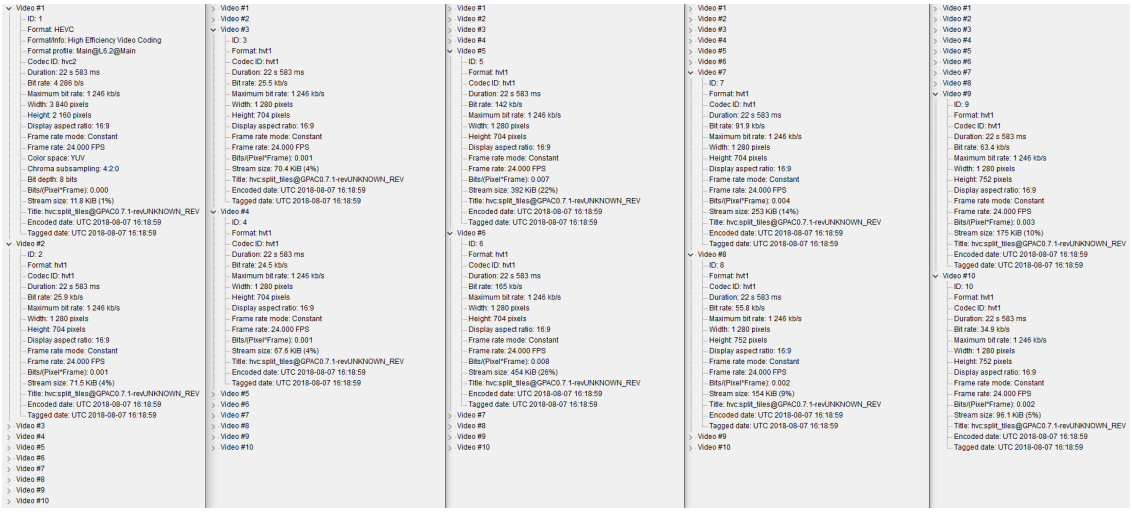


Figure 5.3: The MP4 container with for every tile a different stream and Video 1 containing only non VCL NAL units with the complete video info

The previous two steps have to be done for all videos in all the available qualities. For this example, the qualities 300, 600, 1200 and 2400 kilo bit per second are created. The last step is shown in Listing 5.7, which will combine the different video qualities in a MPD and segment the video files correctly. The output exists in two parts, one can be seen in Section 3.4.1. This is a generated MPD file containing the needed SRD info. The second output are all the tiled HEVC video segments and initialization MP4 for the MPEG-DASH stream.

```
$ MP4Box -dash 1000 -frag 1000 -rap -segment-name %s_segment -min-buffer
     1000 -url-template -out dash_frametile/output_mpd_file.mpd 3840
    x2160_fps24_frametile_300000.mp4 3840x2160_fps24_frametile_600000.mp4
     3840x2160_fps24_frametile_1200000.mp4 3840
    x2160_fps24_frametile_2400000.mp4
```

Listing 5.7: Generation of the MPEG-DASH files based on the MP4 input

5.2 Native application

When the development of the native application started, a few roads could be chosen. One was working on basis of the native application of *GPAC*, which was a custom application based on *FFmpeg* [GPA18]. Or work with *FFmpeg* which can decode HEVC at the time of writing this thesis, just like the web browser HEVC [FFm18]. Choosing *FFmpeg* might seem weird at first, but there are a few reasons why *FFmpeg* was chosen as the baseline. *FFmpeg* is an open source project written in the programming language C which can support a wide variety of codecs. One of those is HEVC which we need. At the time it seemed interesting to use *FFmpeg* because of the many integrations of *FFmpeg* in other products, for example VLC media player by making use the *libavcodec* given by *FFmpeg*. The *libavcodec* is a generic coding library containing a number of audio, video, subtitle stream and bit stream filters, decoders and encoders. Another reason for using *FFmpeg* is because of the multiple web browser versions of *FFmpeg* found. The reason for these multiple web browser implementations is because of the C-implementation which is *easy* to convert with emscripten, more details about this in Section 5.3.

Everything just mentioned were reasons enough to start developing in *FFmpeg* version 3.4.0 to create my own single tiled HEVC video in a MP4 container based on the input and decode this to a working video. However, as just mentioned, *FFmpeg* is not well documented in it is current state, a lot of information is outdated which is understandable in such a huge project but a pain to start your own implementation based on the enormous library. The best way to start a project with *FFmpeg* is to look at the examples given and a debugger. This is what I did and the basis of section 5.2.1.

To make the following explanations easier, which go over the different phases of the master thesis implementation, the input file is always a 3x3 tiled HEVC video, MPEG-DASH segmented or a single MP4 file with one stream per MP4 track for every tile. The input content is generated with the steps from Section 5.1.

5.2.1 Phase 1 - Remux multi stream tiled HEVC MP4 to single stream native tiled HEVC MP4

Phase one is conceptually easy to understand and can be seen in Figure 5.4 in 3 simple steps:

- Input: a single MP4 file containing the tiled 3x3 video in the different streams.
- Remux[1] the input to a MP4 file with a single stream containing one tile per slice.

[1]Remuxing in the video context is an action where one file container is transformed by combining media streams into the new container [Ple18]. This can be the same container or for example from MP4 to MKV. In our context we will remux from multiple video streams to one within the same MP4 container.

- Output: the HEVC video with a single stream in a MP4 container so the video can be tested by playing it in the Microsoft Edge web browser.

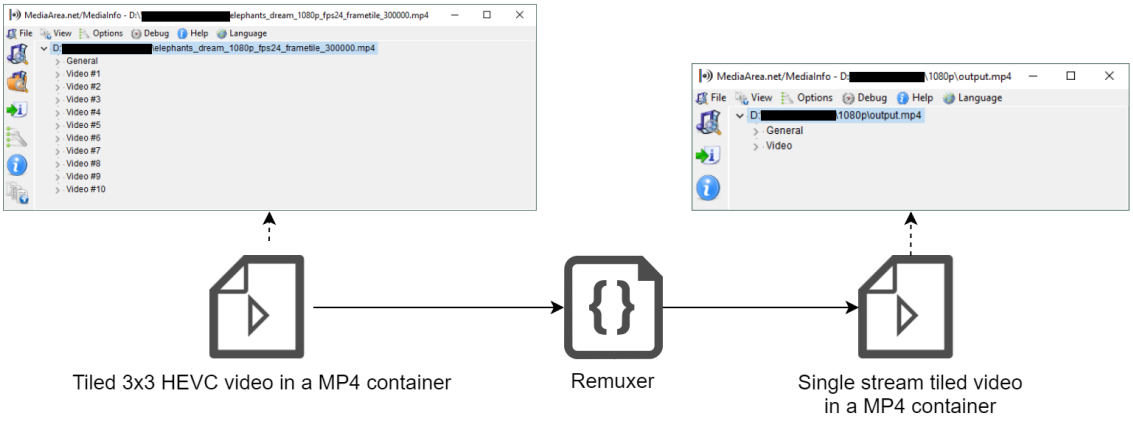


Figure 5.4: Phase 1 program flow from one MP4 containing a 3x3 tiled HEVC video to a single stream HEVC tiled video

To begin remuxing the input file, we need to open the file. This can be done with the code in listing 5.9 on line 2. Only problem with this function is the way it works. First it will define the context based on the file extensions. As the input file is a MP4 file, the extension will be .mp4 which is not a problem so far. Once the file extension has been defined, *FFmpeg* will automatically check the streams in the MP4 to make sure it can work with this file. By default the streams are defined as *hvc2* and *hvt1* as shown in Section 5.1.1. The *hvc2* code stands for a MP4 containing multiple tiles and the *hvt1* represents a tile stream. This was later also verified by the paper [CFD+17]. This was one of the first and worst encounters of the bad documentation of *FFmpeg*. As *FFmpeg* does not know what to do with the codec identifiers *hvc2* and *hvt1*, it refuses to read the streams. An apparently easy but hard to find fix for this problem was found by adding the following two rules defined in Listing 5.8 to the isom.c file, found in the *libavformat* library. Now *FFmpeg* will interpreter the codec identifiers *hvc2* and *hvt1* as HEVC streams.

```
1 { AV_CODEC_ID_HEVC, MKTAG('h', 'v', 'c', '2') }, /* HEVC/H.265 coded
      with gpac */
2 { AV_CODEC_ID_HEVC, MKTAG('h', 'v', 't', '1') }, /* HEVC/H.265 coded
      with gpac */
```

Listing 5.8: Two *rules* to make sure FFmpeg understands and knows how to handle streams with hvc2 and hvt1 encoding.

Once the file has correctly been open and can be read, a new output context is created. This output context will be created based on the extension give by the `out_filename` parameter and is allocated on on line 8.

Now that the output context has been created, it has to include an output stream. The output stream will contain the exact same metadata as input stream 0. This is because of the information contained in stream 0 which is the complete HEVC video information (resolution and bit rate of all tiles combined). For a refresh of the structure of the input data, have a look at Section 5.1.1.

```
1 //Open the input file
2 if ((ret = avformat_open_input(&ifmt_ctx, in_filename, 0, 0)) < 0) {
3   fprintf(stderr, "Could not open input file '%s'", in_filename);
```

```
4     goto end;
5   }
6
7   //Create an ouput context
8   avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL, out_filename);
9   if (!ofmt_ctx) {
10    fprintf(stderr, "Could not create output context\n");
11    ret = AVERROR_UNKNOWN;
12    goto end;
13  }
14
15  //Create a output stream
16  AVStream *out_stream;
17  AVStream *in_stream = ifmt_ctx->streams[0];
18  AVCodecParameters *in_codecpar = in_stream->codecpar;
19
20  out_stream = avformat_new_stream(ofmt_ctx, NULL);
21  if (!out_stream) {
22    fprintf(stderr, "Failed allocating output stream\n");
23    ret = AVERROR_UNKNOWN;
24    goto end;
25  }
26
27  ret = avcodec_parameters_copy(out_stream->codecpar, in_codecpar);
28  if (ret < 0) {
29    fprintf(stderr, "Failed to copy codec parameters\n");
30    goto end;
31  }
32  out_stream->codecpar->codec_tag = 0;
```

Listing 5.9: Opening the input file into an input context object

In *FFmpeg* the term *AVPacket* or packet is used for a frame that is still encoded, the term *AVFrame* is a frame that is decoded. The same terms will be used in this explenation. *FFmpeg* works with a frame reader which will read every frame of the given input context in sequential order. However, this means there is no control of what package should be read in what order. *FFmpeg* chose to implement the packet reader this way because of optimizations. If they did allow for a search in the reading of a packet, in worst case a complete file had to be read to find one specific packet which would take too much time in some cases. This does however have the drawback that the complete file will be read in sequence and a if-test has to be done if this is the correct packet and what to do with it. Take for example our case where every stream is grouped together in groups of X packets of every stream. This means that for every sequence of packets first X amount of stream 0 will be read, then x amount of packets of stream 1 continuing this situation till stream 9. Let us give the hypothetical situation where the implementation would work by first searching for a first packet of stream 0, then a packet of steam 1 till frame N+1 (with N tiles). Once this is done, combine these packets and search for the next series of corresponding packets for each of the input streams. This means that as the streams are not in order (which is the case in our situation), a search has to be done throughout the complete file every time a packet has to be combined which is suboptimal.

Listing 5.10 will show how the packets are ordered by stream. By requesting the amount of packets in the stream, an 2D-array can be allocated that will contain all packets of the stream. The first index of the array will be the corresponding stream index while the second index will be the packet number. Once the array is allocated, the file will be read in a sequential order with a while loop. This loop will continue to read each packet

until no more packets are found. Each packet is copied with line 6 into the correct array location.

```
1 while (1) {
2   pkt = (AVPacket*)malloc(sizeof(AVPacket));
3   ret = av_read_frame(ifmt_ctx, pkt);
4   if (ret < 0)
5     break;
6   av_copy_packet(&arrayOfStreams[pkt->stream_index][packetIndexArray[
        pkt->stream_index]] , pkt);
7   packetIndexArray[pkt->stream_index]++;
8 }
```

Listing 5.10: Sequential loop through the complete file while saving each stream in there corresponding array index

Once all packets have been read, the last two steps of the process can begin. The first will be the combination of the read packets, secondly the packets have to be written to the output stream. This is done with the code in Listing 5.11. On line 2 the combined frame size will be calculated based on the packets retrieved in previous step. Once the size of the combined packet is known, the new packet data buffer is allocated. Once this buffer is allocated, all the data is being copied from the different packages to the data buffer. After the data has been copied, a new *AVPackage* is created based on the new data on line 16. When the new packet has been made, it has to be filled in with meta info such as duration, PTS, DTS and to what stream it has to be written. Now that the packet has been completely prepared, it is ready to be written to the correct stream, this is done on line 26. After writing the packet to the output stream, only cleaning up the memory is needed. These steps have to be repeated for all output packets.

Another important thing is the way the `AVPackets` are structured and the need to use the correct buffer size for the `memcpy` on line 10. On line 10 the packet data from the buffer gets copied to the new packet. However, the size of the packet is requested with *tempArray[streamIndex].size* which and not the size of the buffer. This is because when requesting the size of the buffer, there will be 32 extra bits added. These are the parameters of the *AVPacket* while only the data is being copied.

```
1  for(int column = 0; column < nb_frames; column++){
2    bufferSize = calculateBufferSize(arrayOfStreams, column);
3
4    //Create uint8_t which will hold the different tile(s) data
5    uint8_t *data = (uint8_t *)malloc( buffSize * sizeof(uint8_t) );
6
7    //Copy the frame data to the new combined frame on the correct
         location
8    int index = 0;
9    for(int streamIndex = 0; streamIndex < stream_mapping_size;
         streamIndex++){
10     memcpy( data + (index * sizeof( uint8_t)),
11           tempArray[streamIndex].buf->data,
12           tempArray[streamIndex].size * sizeof( uint8_t));
13     index += tempArray[streamIndex].size * sizeof( uint8_t);
14   }
15
16   //Create a packet from the data
17   int ret = 0;
18   if((ret = av_packet_from_data(&combinedPkt, data, buffSize)) < 0){
19     fprintf(stderr, "Welp\n");
```

```
20    }
21
22    // copy packet info of stream 0 to the new created packet
23    ...
24
25    //Write the new combined frame to the output stream
26    ret = av_interleaved_write_frame(ofmt_ctx, &combinedPkt);
27    if (ret < 0) {
28      fprintf(stderr, "Error muxing packet\n");
29      break;
30    }
31
32    //Clean memory
33    ...
34  }
```

Listing 5.11: Combine the tile packets and then write them to the correct output stream

Only thing left is writing the MP4 trailer by using the `av_write_trailer(ofmt_ctx)` command after all packets have been processed and closing all contexts while freeing the leftover memory.

5.2.2 Phase 2 - Remux DASH encoded tiled HEVC stream to single native tiled HEVC dash stream

As the input of this phase is an *init.mp4* file with 10 segments (each representing a tile track besides track 1 which are the NAL units needed to decode the tiles as described in Section 5.1) an approach had to be found to combine these files to an input that could be used to process and eventually generate the wanted output. An even better situation would be if the steps from phase 1 could be reused.

By looking into ISO/IEC23009-1:2014 [ISO14b], it became clear that segments can in some cases be concatenated with each other to form a valid bit stream. The concatenation of the files goes as follows: the *init.mp4* file is placed as the first file, and then all media segments get concatenated behind each other in any order behind the *init.mp4* file.

Once the new concatenated file has been made, the same steps as in Phase 1 can be followed to generate a single stream tiled MP4 video. A visual representation can be seen in Figure 5.5 as the concatenation is now part of the *remuxer*.

Now that the process from segments to a native tiled single stream MP4 is working, the output to MPEG-DASH segments is needed. There are two major reasons for this. The first one being a web browser restriction as the end product is meant to be a web application. The end product has to be a video stream in which the user can seek for different timestamps with segments as input. If the output would be a MP4 file, only a part of the video would be played (in case of a video with multiple segments per tile which is mostly the case) and reinitialization of the video element has to be done when adding the new MP4 generated from the segments. This would eventually lead to reinitializing the video element over and over with a new MP4. Another reason is the media source buffer that will be used in the *JavaScript* implementation. By using the media source buffer, adaptive streaming is possible. A client can choose the qualities as needed and

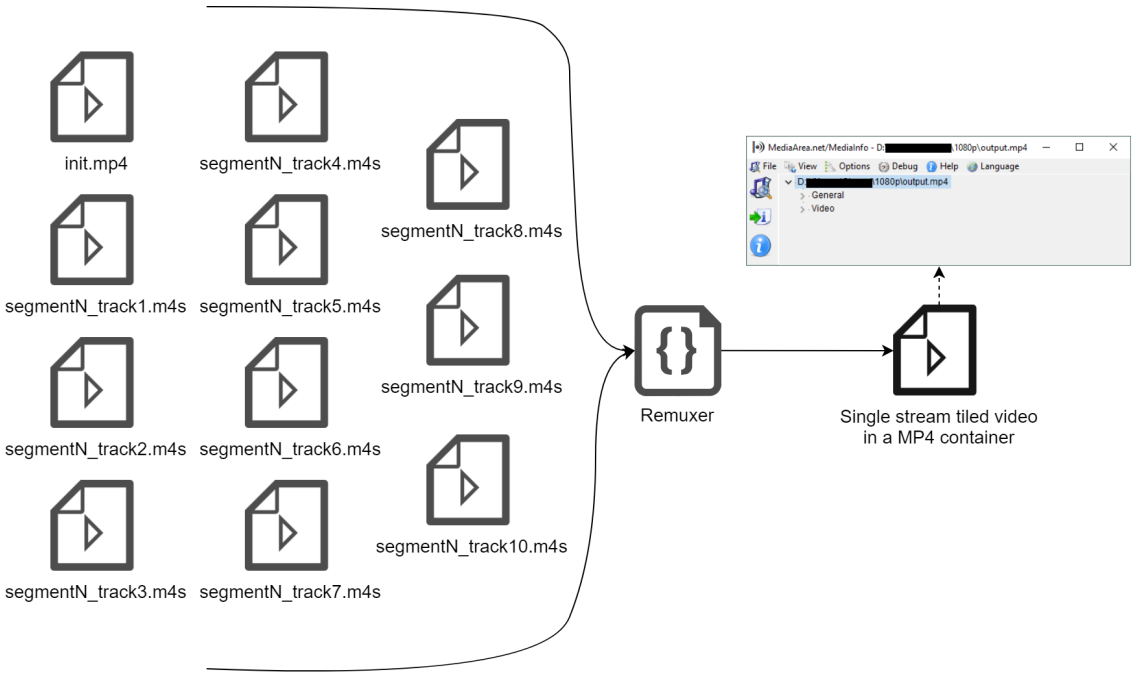


Figure 5.5: Phase 2 program flow from segments containing a 3x3 tiled HEVC video to a single native tiled HEVC video

request these qualities. Because of these reasons, the choice was made to generate MPEG-DASH-like output. A detailed view of how the output generated in this step is processed for playback in the web browser can be read in Section 5.3.

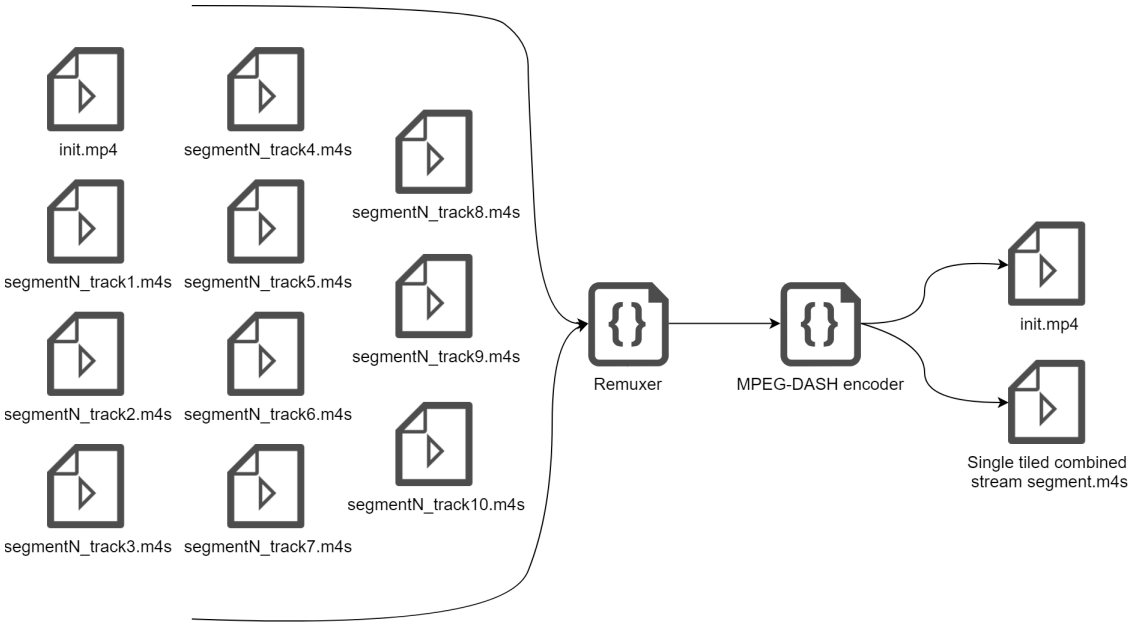


Figure 5.6: Phase 2 program flow from segments containing a 3x3 tiled HEVC video to a single native tiled HEVC MPEG-DASH stream

As can be seen in Figure 5.6, a new step has been introduced, the *MPEG-DASH encoder* step. A first version of this implementation took a MP4 file generated by the previous step as input and reformed this into a MPEG-DASH stream. However, this generated

two extra files, the concatenated file and the intermediate native tiled single stream MP4 file created from the segments. This single stream MP4 would then be converted to a DASH-stream with only one segment. Eventually the implementation would be converted to a web application so in the final version, the step to create a MP4 has been removed as generating more files would mean more RAM usage in the browser, something that has to be avoided if the application would also be used on low RAM devices.

The generation of the MPEG-DASH output will based on *FFmpeg* code. By using *FFmpeg* for all steps only one program has to be converted to the web application. An alternative in our processing chain, could of been the tool provided by *GPAC*: MP4Box. However, this would mean a second program has to be converted to *JavaScript*. A detailed explanation of the *JavaScript* workings can be found in Section 5.3.

Just as with the generation of the output stream in the previous phase, an output stream has to be initialized. This is done with two functions: `init_dash_context` and `dash_init`. An important difference with phase 1 is that the input context will also be the output context because of the way the *DashEncoder* of *FFmpeg* has been implemented. The initialization method `init_dash_context` has been implemented by myself and is needed because the *DashEncoder* by *FFmpeg* is a single program that takes variable set by arguments in command line. These variables will always be the same for our situation. If they would also be set by the command line, the input of the program would be the output file name, initialization file name, tile segments and all arguments for the *DashEncoder*, which would be the same every execution. The variables that have to be set are:

- *init_seg_name* this parameter defines the name of the output initialization MP4 file.
- *media_seg_name* this parameter is the name of the single output segment containing the native tiled HEVC video.
- *adaptation_sets* this parameter is *'id=0,streams=0'* and represents the stream that should be mapped on the id of one adaptation set. This means that one **AdaptationSet** exists which contains stream 0.

When the initialization of the command line arguments is done, the original `dash_init` method has to be called. However, the original implementation of the *DashEncoder* at the moment of writing of this master thesis (*FFmpeg* version 3.1.9001) worked with either one file which would output the initialization and segments files together in one file or generate a individual file for all outputs, meaning that there will be a initialization file and a segment file for every frame of the video. These two outputs were both not interesting in our situation. The output we needed was a *init.mp4* to initialize the video element and one segment that was just as long as the input segments.

The complete overhaul of the *DashEncoder* implementation will not be discussed in detail, only the important parts will be discussed and explained. In the implementation of the *DashEncoder* there will be a check to see if the output container of the initialization file is a MP4 file, if so the *init.mp4* file will not be written until the first packet has been processed and is ready to be written. This is done in the method `dash_flush` which can be seen in Listing 5.12. *dash_flush* is a method to flush the DASH-stream. On line 5 there is a check to see if the initialization of the output stream has been written by checking the size of the init.mp4 file. If this is 0, the file has yet to be written for the specified output context.

Once the initialization file has been written, the new packet that has to be added to the

segment get added with all parameters on line 25. Furthermore, the data buffer gets flushed on line 36. If nothing has been flushed or something went wrong, a break will happen on line 38. This is because all discussed parts of the `dash_flush` method are part of a for-loop that handles all output streams. In our case we only have one output stream as the files are being written in a single native tiled output stream. In Listing 5.11, line 26 will be replaced with `ret = dash_write_packet(ifmt_ctx, &combinedPkt)` to call the dash write function.

```
1  ... Init variable
2
3  // Check if the initialization MP4 file has to be written as this
4  // might be the first packet to be written
5  if (!os->init_range_length) {
6    flush_init_segment(s, os);
7    ret = dash_setup_seg_file(s);
8    if (ret < 0) {
9      fprintf(stderr, "Error while setting up dash_setup_seg_file\n");
10   }
11 }
12
13 // Initialization of the  the destination of the media segment string
14 snprintf(full_path,
15           sizeof(full_path),
16           "%s%s",
17           c->dirname,
18           c->media_seg_name);
19
20 // Find the location of the last written segment
21 find_index_range(s, full_path, os->pos, &index_length);
22
23 // Add the new packet to be written to the media
24 // file and write it to the media file
25 add_segment(os, //OutputStreamDashEnc *os
26             full_path, //const char *file
27             os->start_pts, //int64_t time
28             os->max_pts - os->start_pts, //int duration
29             os->pos, // int64_t start_pos
30             range_length, //int64_t range_length
31             index_length); //int64_t index_length
32 os->pos += range_length;
33
34 // Flush any residue packets from the buffer in case we are
35 // working with multiple streams for example
36 ret = flush_dynbuf(os, &range_length);
37 if (ret < 0)
38   break;
39 os->packets_written = 0;
40
41 ... Write to manifest, not the case in this implementation
```

Listing 5.12: Implementation of the dash_flush method to write all packets to the relevant files

Just like in phase 1 needs the output context and memory be freed correctly. As an extra, the temporary concatenated file which was created at the start of this phase has to be removed as this file has no more purpose.

This phase is also the phase that has been implemented in the web browser. More information on the reasons why can be read in the next phase and Section 5.3.

5.2.3 Phase 3 - Decoding DASH encoded tiled HEVC stream to JPEG images

A good use case seemed to try to decode the video into raw YUV format or JPEG. This would allow other web browsers then Microsoft Edge and Apple Safari to decode the HEVC video as these two are the only two browsers with HEVC support at the time of writing this thesis. By decoding into raw YUV format or JPEG images, a web browser canvas element could display these images in a timely matter which would give the same results as *playing* the video in a video element. After implementing this as a native application, it became very clear that theoretically this seems as an interesting use case, yet the practical experience was less than optimal as will be further explained in this section.

Just as in phase 2, everything from phase 1 can be reused with a few modifications. Instead of calling line 26 in Listing 5.11, the pseudo-code in Listing 5.13 will be called. In this Listing, the `video_dec_ctx` is assumed to already be created and ready to decode HEVC content to YUV raw image. The initialization of the decoder can be done with `dec = avcodec_find_decoder(AV_CODEC_ID_HEVC)` to check if the decoder exists and the initialization of the context with `video_dec_ctx = avcodec_alloc_context3(dec)`. On line 2 of Listing 5.13, the packet is decoded to `frame`. If the decoding of the packet went well, the new decoded frame can be saved as a JPEG with the function call on line 13.

```
1  ... Initialization
2  ret = avcodec_decode_video2(video_dec_ctx, frame, got_frame, pkt);
3  if (ret < 0) {
4    fprintf(stderr, "Error decoding video frame (%s)\n", av_err2str(ret))
5    return ret;
6  }
7
8  if (*got_frame) {
9    ... check if correct frame has been generated
10 }
11
12 // Save frame as jpeg
13 save_frame_as_jpeg(video_dec_ctx, frame, *video_frame_count);
```

Listing 5.13: Pseudo code for the decode packet implementation

Once `save_frame_as_jpeg` has been called, whose implementation can be seen in Listing 5.14, the JPEG encoder has to be made to save the frame. The check to see if the JPEG encoder is available and creation of the decoder can be seen on line 3, 7 and 17. When the context has been created, a few output settings have to be made to minimize the size of the output JPEG file; these settings can be seen on line 10-14.

Now that the encoder has been created, the `AVFrame` has to be encoded. As discussed before, an encoded frame is an `AVPacket`, as such one is initialized on line 24. On line 28 the actual encoding of the frame will be done and then saved on line 33.

After correctly closing and removing memory, the recoding to images in seperate files is done.

```
1  static int save_frame_as_jpeg(AVCodecContext *pCodecCtx, AVFrame *
       pFrame, int FrameNo) {
2    // Check if the encoder exists
3    AVCodec *jpegCodec = avcodec_find_encoder(AV_CODEC_ID_MJPEG);
```

```
4   if (!jpegCodec) { return -1; }
5
6   // Allocate the JPEG context
7   AVCodecContext *jpegContext = avcodec_alloc_context3(jpegCodec);
8   if (!jpegContext) { return -1; }
9
10  jpegContext->pix_fmt = AV_PIX_FMT_YUVJ420P;
11  jpegContext->height = pFrame->height;
12  jpegContext->width = pFrame->width;
13  jpegContext->time_base = pCodecCtx->time_base;
14  jpegContext->qcompress = 0.1f;
15
16  // Open the codec with the correct parameters
17  if (avcodec_open2(jpegContext, jpegCodec, NULL) < 0) { return -1; }
18
19  FILE *JPEGFile;
20  char JPEGFName[256];
21
22  // init AVPacket
23  AVPacket packet = {.data = NULL, .size = 0};
24  av_init_packet(&packet);
25  int gotFrame;
26
27  //Encode the frame as an JPEG image
28  if (avcodec_encode_video2(jpegContext, &packet, pFrame, &gotFrame) <
        0)   { return -1; }
29
30  sprintf(JPEGFName, "dvr-%06d.jpg", FrameNo);
31  //write the decoded frame to a file
32  JPEGFile = fopen(JPEGFName, "wb");
33  fwrite(packet.data, 1, packet.size, JPEGFile);
34  fclose(JPEGFile);
35
36  ... free memory
37  return 0;
38 }
```

Listing 5.14: Saving a decoded frame to file as a JPEG image

As mentioned before, this implementation is not suitable for web browser contexts. There are two major reasons for this, one is the HEVC decoding and then encoding process that would take too much time. The second problem is the size of the decoded YUV frames, these frames would take up a lot of RAM usage. This is also the reason why the choice to encode to JPEG images was made. Saving a YUV to eventually play in a canvas element would take enormous amounts of RAM as the web worker has no direct access to save to user memory. Even a single YUV frame size is too big and halts the web browser from working.

5.3 JavaScript Implementation

After completing the C-implementation, described in Section 5.2, the conversion to *JavaScript* was the next step. A complete overview of what steps will be taken in *JavaScript* can be seen in Figure 5.7. First the topic of how the conversion was made from C code to a *JavaScript* implementation will be handled. Then the complete workings of the actual web browser application will be discussed. The web application has been written in

ECMAScript 6 JavaScript.

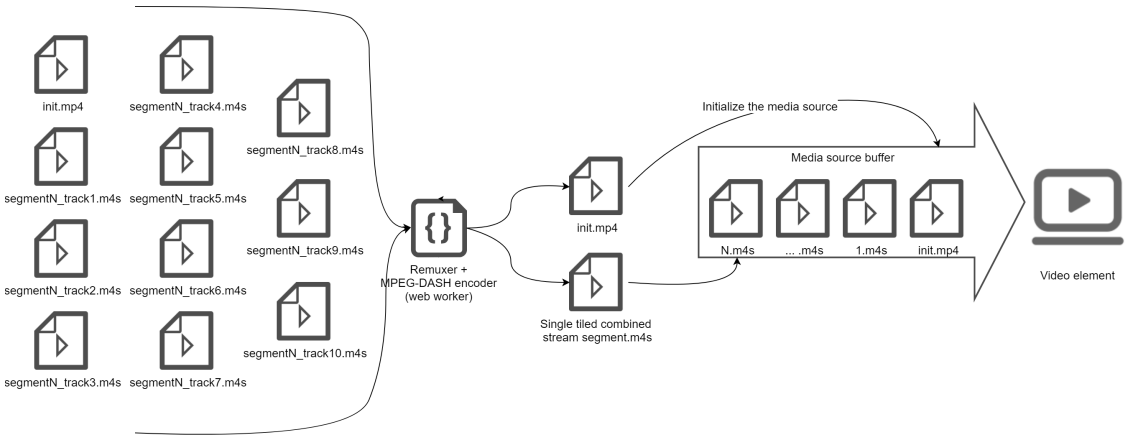


Figure 5.7: Complete overview of the flow of the JavaScript implementation

5.3.1 Conversion of the native application to JavaScript

At the start there seemed two possible options to convert the C-implementation of phase 2 in Section 5.2: either cut the bit stream in pieces myself by reading HEVC video bit stream headers in *JavaScript*, or try to convert the *FFmpeg* implementation to a *JavaScript* implementation with tools. The latter approach was taken as there were cases found on *GitHub* of *FFmpeg* in *JavaScript*. However, most cases were a simplified version of *FFmpeg* which did not have all options included. After some research, more info about web assembly was gained and how it could be used with conversion scripts to transpile my *FFmpeg* implementation of Phase 2 to *JavaScript*.

The conversion of the C-implementation of the code to *JavaScript* is based on the scripts provided by *Kagami* [Kag18]. *Kagami* provides scripts which will use *Emscripten* [ems18] to convert our *FFmpeg* version 3.1.9001 to a working *JavaScript* version with restrictions (e.g., only allowing to work with MP4 containers and a specific codec). By updating the scripts provided by *Kagami* to include only the used video containers and codecs, and using the *FFmpeg* version of the phase 2 implementation, a conversion was possible. It is important to note that the correct *Emscripten* version is used: 1.37.40 this is because the latest version of *Emscripten* by default generates WASM [web18b] while we need ASM.JS. ASM.JS is a very strict subset of *JavaScript* and has a lot of C characteristics [Bam18]. It is not needed to go in complete detail of ASM.JS in this master thesis but it is important to understand that the eventual conversion of the C-implementation yields ASM.JS code which is very optimized *JavaScript*.

The converted *JavaScript* implementation works based of *web workers*. Web workers allow a web browser to run scripts on another thread in the background without interfering with user interaction [web18a]. A web worker can be accessed by sending post messages and getting messages back with the *onmessage* event handler which can be seen in Listing 5.15. On line 2, the web worker is created. On lines 5 to 23, all message cases get handled. In the situation of our conversion, the response can be

- *ready* when the web worker is finished loading.

- *stdout*, *stderr*, *exit*, *error* are messages coming from the C-implementation which are discarded in this case.
- *done* means that the web worker is finished with the processing of the input.

Lastly there is the `postMessage`, seen on line 27, which allows for starting the web worker with a set of arguments.

- The attribute *type* says the web worker should execute the program with given parameters.
- *MEMFS* is a virtual file system used by *Emscripten* to allow input/output files to be exchanged between the invoking thread and the web worker. It is important to notice that when the web page gets refreshed, all content will be removed.
- *arguments* is a list of strings to be used as arguments just like in a native application with command line execution.

```
1  // Load the web worker with a script
2  var worker = new Worker("segmenter/ffmpeg-worker.js");
3
4  // What to do if the worker has a message
5  worker.onmessage = function(e) {
6    var msg = e.data;
7    switch (msg.type) {
8      case "ready":
9        break;
10     case "stdout":
11       break;
12     case "stderr":
13       break;
14     case "exit":
15       break;
16     case "error":
17       break;
18     case "done":
19       var outputInit = msg.data.MEMFS[1];
20       var outputSegment = msg.data.MEMFS[2];
21       ... if needed add init file to media source buffer (first segment
             )
22       ... else add only media segment to media source buffer
23       break;
24     }
25 };
26 //start the web worker
27 worker.postMessage({type:"run", MEMFS: segmentList, arguments: [...]});
```

Listing 5.15: Web worker example implementation

One of the restrictions of working with the web worker implementation is that all segments and the initialization MP4 have to be downloaded and saved in an *ArrayBuffer* to enable them to be input into the web worker. To cope with this situation, the amount of tiles is read from the MPD. Once we know how many tiles are needed, the corresponding segments are downloaded via the *scheduler*, more details about the *scheduler* will be given later on. When a segment is received, it will be placed in the *segmentList* by creating an object which contains two attributes: *name* and *data*. The name is the segment with the corresponding tile number, the data is the ArrayBuffer. Once all tiles have been received and if needed the *init.mp4* file, the web worker receives the *segmentList* as input with the

arguments list, which is a list of all segment names and the *init* filename. The *init.mp4* is needed every run, as such this file can be cached and reused instead of re-downloading the file with the segments.

When the web worker finishes, the *done* message gets executed on line 18 of Listing 5.15. It will load the created init and segment file out of the `MEMFS` memory and add the init file (if needed) and segment file to the media source buffer. The Media source buffer will be explained in Section 5.3.2.

5.3.2 Web application

Now that the conversion of the native application is done and the messaging from and to the web worker is understood, the next step is including the web worker in the web application written in EmcaScript 6. EmcaScript 6 or ES6 is a new version of *JavaScript* that includes new features such as classes and is supported by all major browsers with the latest version. For a complete overview of all new features, have a look at [Eng].

The web application works by using a video element, the media source and the media source buffer. First the media source gets initialized on line 2 of Listing 5.16 which will eventually (line 5) act as a source for the video element. A event listener which checks if the source is open gets attached on line 3. Once the source is open, the function on line 8 is called. In this function the to use video codec (which is extracted from the MPD file, in our case this will he HEVC) will be initialized. This video codec is read from the MPD file which has been requested by a HTTP GET request earlier.

```
1  initSourceBuffer() {
2      this._mse = new MediaSource();
3      this._mse.addEventListener('sourceopen',
4          onSourceOpen.bind(null, this._videoElement, this._mse));
5      this._videoElement.src = URL.createObjectURL(this._mse);
6      let videoController = this;
7
8      function onSourceOpen(video, mse, evt) {
9          Logger.debug(VideoPlayer.getTag(), "onSourceOpen()");
10         try {
11             videoController.sb = mse.addSourceBuffer(
12                 videoController._videoCodec);
13             Logger.debug(VideoPlayer.getTag(), 'source buffer added');
14         } catch (Exception) {
15             Logger.error(VideoPlayer.getTag(),
16             "Could not add the specified video codec");
17         }
18     };
19 }
```

Listing 5.16: Initialization of the video element with a media source buffer to input segments

Adding media segments to the media source buffer is done in Listing 5.17 with a self implemented queue. This had to be done because of a restriction in Microsoft Edge which would not allow the adding of the event handler *onupdateend*. This event handler would be called each time the buffer was done updating and could then be asked to push the new element into the buffer. Because Microsoft Edge did not allow this, a queue that works based on promises was developed. *Promises* have been around for a while in different

libraries but have only recently been added to the main library of JavaScript [Arc18]. A *promise* allows for an asynchronous execution that will return either with a resolved value or a reject. The queue works by seeing the next item in the queue (a function call) as a resolve and executing the next item in the queue. To counter the disability of Edge to use the event handler *onupdateend*, an interval is set for each segment that has to be added to the `SourceBuffer`. These intervals will be added to the queue and executed in order. When an interval is successfully completed, the next interval in the queue will start to add the next segment.

As can also be seen in Listing 5.17 on line 5, there is a check to see if the source buffer already holds data. If this is the case, the last buffer time stamp is taken and the new segment needs to be appended on this time stamp. This approach had to be taken because of a problem with the *DashEncoder* implementation of phase 2 in Section 5.2.2. Within this implementation, the time stamps are reset every new run. This means that when a new segment is processed, the time information always starts from zero. A bug fix for this has not been found in the current implementation because of the lack of documentation by *FFmpeg*. For this reason the *JavaScript* side bug fix has been introduced.

```
1  addToMediaSourceBuffer(arrayBuffer) {
2      let sourceBuffer = this.sb;
3      this._mseAppendQueue.add(function () {
4          let appendFunction = setInterval(function () {
5              if (sourceBuffer.updating === false) {
6                  if (sourceBuffer.buffered.length !== 0) {
7                      sourceBuffer.timestampOffset =
8                        sourceBuffer.buffered.end(0);
9                  }
10                 sourceBuffer.appendBuffer(arrayBuffer);
11                 clearInterval(appendFunction);
12             }
13         }, 10);
14     })
15 }
```

Listing 5.17: Adding of the video data to the media source buffer by queue because of Microsoft Edge restrictions

The last important piece of the web application is the *scheduler*. As mentioned before, the scheduler works on basis of a *steady fill* implementation. This means that when the buffer notices it does not have enough video data buffered based on the minimum buffer time and current playback time, the next segment gets requested. However, Microsoft Edge has a restriction wherein the media source buffer has to have at least 5 seconds of video content before the playback of the video is possible. Therefor, has the implementation been made to first download and remux 5 seconds of video content and then use the minimum buffer time to check if the next segments should be downloaded or wait. The *scheduler* has been constructed in such a way that another buffer fill algorithm can be added by inheriting from *ITiledDownloadScheduler* and implementing an algorithm to calculate what qualities should be used. In our example as shown in Figure 5.8, the quality management is straightforward. The *viewport* which can be seen on the left side can be moved across the whole video on the right side. It is also possible to make the viewport bigger and smaller by scrolling. The tile that has the most surface in the viewport will be given the best quality available, all other tiles will get the lowest quality.

The web application also allows to update the quality of a tile manually by selecting a

quality from the 'bit rate per tile' list. This list goes from tile 1 until tile 9 in this case. Each tile has a possibility of 4 qualities as 4 MPEG-DASH representations are available. When moving the viewport, this quality does not change unless the tile has the most surface in the viewport, which will make the tile automatically update to the maximum bitrate.

It is also possible to pause the video. When an user pauses the video, the scheduler will halt the downloading of new segments as the current play time of the video does not change, implying that the buffered content will not decrease nor increase.

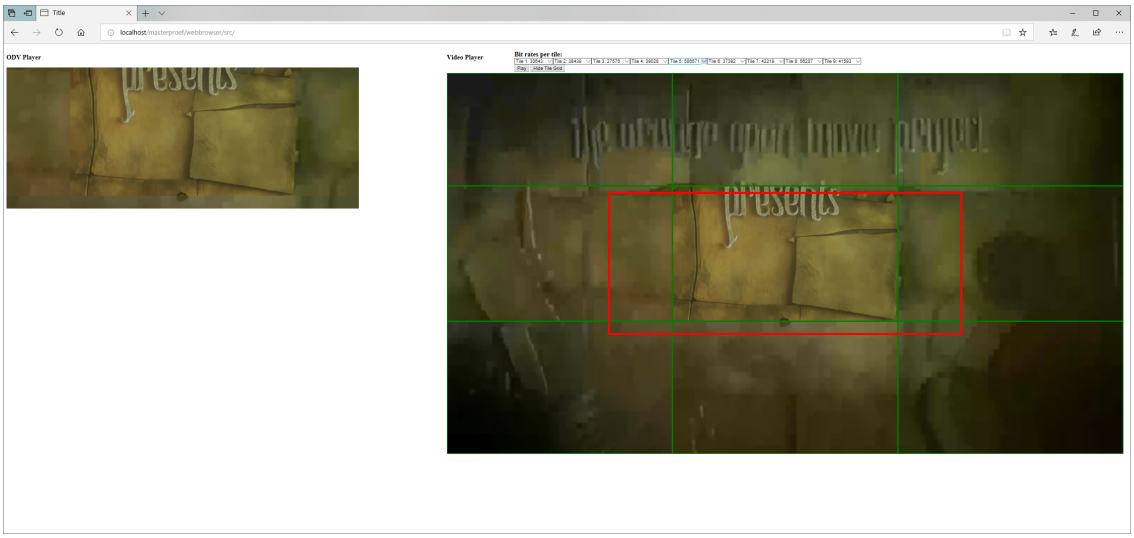


Figure 5.8: The web application as seen in the web browser

Chapter 6

Results

In this chapter, the results will be analyzed by first describing the setup with which the tests were conducted. In this use case the test are done on a Windows desktop for the Microsoft Edge, Google Chrome and Mozilla Firefox web browsers. The web application has been tested on an Apple device in the Safari web browser and verified as working. However, no test results from the execution in the Apple Safari web browser will be used in this analyses. This is because no comparison between the Apple device and the Windows device can be made. The Apple device that was available during the test had nowhere near the same hardware specifications as the Windows device. Another problem was the workings of Google Chrome on the Apple device. There was no knowledge on the operating system interactions Google Chrome made on an Apple device and if these were the same on a Windows device. Therefore, no comparison could be made.

Once the setup is discussed, different resolution and *tile* setup videos are tested in comparison of each other. The conducted tests are available in Section 6.2.

After the experimental results in Section 6.2, a deeper analyses of the web worker execution is done in Section 6.3.

6.1 Setup

The test were made using a windows desktop with the following specifications:

- CPU: Intel® Core™ i7-8700K Processor (3.70 GHz turbo boost to 4.70 GHz)
- Motherboard: ASUS ROG strix z370-E Gaming mainboard Socket 1151
- GPU: NVidia GTX780 3072 MB
- RAM: 16GB DDR4 2.666Mhz
- OS version: Windows 10 Pro 64 bit version 1803 (64 bit)

The web browser version used were:

- Microsoft Edge: 42.17134.1.0 (64-bit)
- Google Chrome: 68.0.3440.106 (Official Build) (64-bit)
- Mozilla Firefox: 61.0.2 (64-bit)

It was of utmost importance to know if every web browser produced the correct output with the web application described in Section 5.3. Therefore, every output of every web browser has been tested and played by the same Microsoft Edge browser on a website containing a simple media source buffer implementation that feeds the produced segments to the media source buffer. Hereby, giving a visual verification of the correct output.

To make sure that the bandwidth does not interfere with the execution time, the assumption is made that there is unlimited bandwidth available. This means that the downloading of segments or any other content needed to produce the *init.mp4* and segment files will not interfere with the execution time in any of the provided web browsers. To give a small reminder, to generate one output segment for a 3x3 tiled video, there are 11 input segments needed. These input segments exist of one initialization MP4 which does not need to be re-downloaded every time and can be cached. The 3x3 + 1 segment have to be downloaded every new segment. For a detailed explanation of the structure of the input, see Chapter 5.

The used input files to test the web application are the following:

- 1920 by 1080 pixels (1920p) 3x3 tiled video of 1000 milliseconds segments.
- 3840 by 2160 pixels (2160p) 3x3 tiled video of 1000 milliseconds segments.
- 1920 by 1080 pixels (1920p) 3x3 tiled video of 5000 milliseconds segments.
- 3840 by 2160 pixels (2160p) 6x6 tiled video of 1000 milliseconds segments.

The 2160p video is 24 seconds long, this means there will be 24 output segments. The 1080p video is 90 seconds long which means 90 output segments for a segment duration of 1000 milliseconds and 18 for the 5000 millisecond duration.

The generation of the output has been done 10 times for each web browser for every test. These 10 executions are then combined into one average execution time per produced output segment. The average is taken to mediate out possible outliers.

6.2 Experimental results

The analysis of the results will go in following steps with a 3x3 tiled video unless stated otherwise. First the execution time of the 1920 by 1080 pixels (1080p) video will be discussed in Section 6.2.1. Then the average execution times of the 3840 by 2160 pixels (2160p) video in Section 6.2.2. Both videos will have segment durations of 1000 milliseconds and will be compared against each other to see the impact of video resolution size in Section 6.2.3. Thereafter, the segment duration will be increased to 5000 milliseconds for the 1080p video and compared with the 1000 millisecond video to see what impact an increased segment duration has on the processing time in Section 6.2.4. Lastly, the tile setup will be changed for the 2160p video from 3x3 tiles to 6x6 and compared with each other in Section 6.2.5.

6.2.1 1080p

In this test, the execution time of the 1000 milliseconds 3x3 tiled 1920x1080 resolution video was tested. To generate one output segment, 11 input files are needed as discussed

before.

When looking at the graph of Figure 6.1, something weird can be seen: the first segment takes much longer than all other segments. At first, we did not think this was a web browser issue. We rather thought it was because of the fact that the first segment, more precisely the first frame, took longer because of the I-frames. This made no sense as all segments started with an I-frame. When looking at the other web browsers, we also notice this spike for Chrome but not for Mozilla Firefox which gave a skewed view in Table 6.1 where the average execution times are shown. The average execution time is taken from the execution times for each individual generation of 90 output segments. The first segment processing time is on average 1400 milliseconds for Microsoft Edge and 907 milliseconds for Google Chrome. In Section 6.3 a deeper analyses for the spike and reason why is examined.

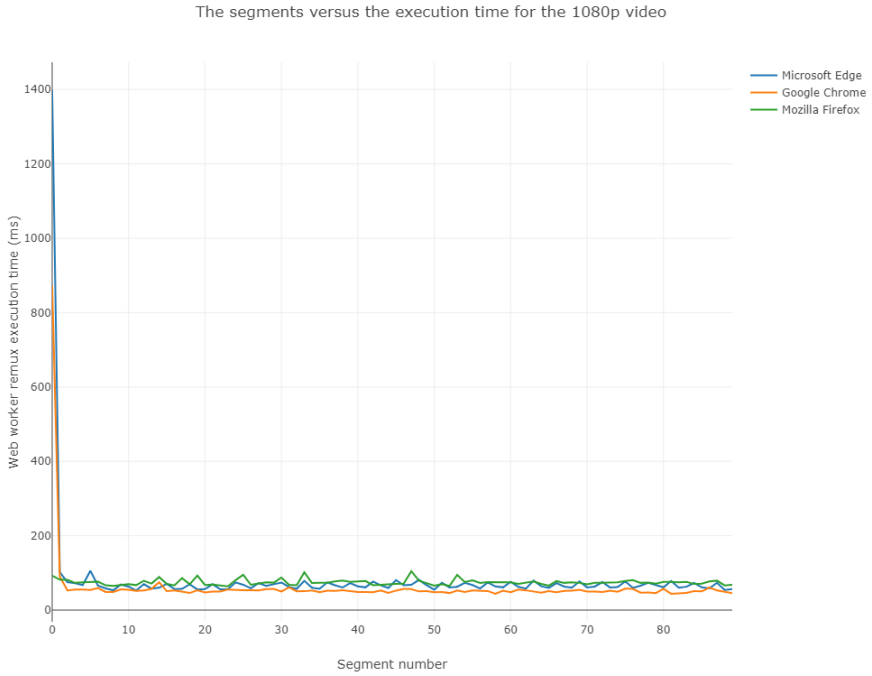


Figure 6.1: Graph showing the execution times of the web worker remuxing step per 1000 millisecond segment for a 1080p video

	1920x1080 pixels (1080p)	
	With first remux	Without first remux
Microsoft Edge	81.205 ms	66.411 ms
Google Chrome	61.221 ms	52.108 ms
Mozilla Firefox	74.817 ms	74.617 ms

Table 6.1: Table showing the average execution times of the remuxing with the web worker for a 1080p video

By removing these first segment times, we get the average execution times shown in Table 6.1 in the column *Without first remux*. The exact timing for each segment other than the first can be seen in the graph of Figure 6.2. In this graph a lot of spikes are seen for a lot of segments in comparison with the previous or subsequent segment, a deeper look into

this, is in Section 6.2.2. Google Chrome seems to be the fastest followed by Microsoft Edge. By looking at table 6.1 we can confirm this statement.

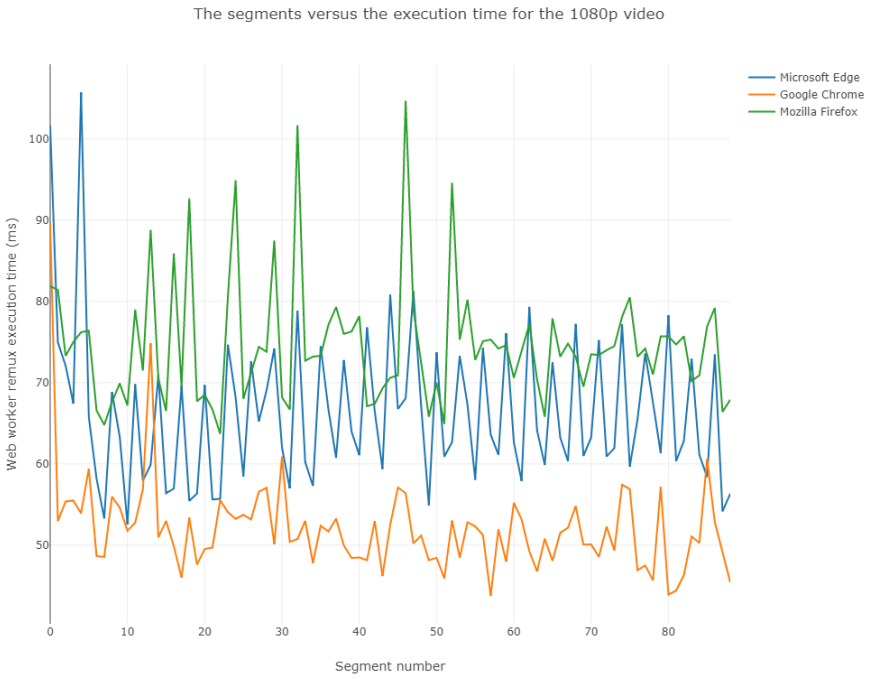


Figure 6.2: Graph showing the execution times of the web worker remuxing step per segment without the first segment for a 1080p video

6.2.2 2160p

In the execution with 3840 by 2160 pixels (2160p) video, the same pattern emerges as in the 1080p video. In the Microsoft Edge and Google Chrome web browser, the execution times of the first segment are extremely high compared with all subsequent executions. The exact execution times for the first segments on average are 1405 and 871 milliseconds for Microsoft Edge and Google Chrome, respectively. This can be seen in the graph of Figure 6.3.

An overview of the average execution times with and without the first segment execution can be seen in Table 6.2 and Figure 6.4. Furthermore, is the average execution time of Google Chrome lower than those of Microsoft Edge and Mozilla Firefox.

	3840x2160 pixels (2160p)	
	With first remux	Without first remux
Microsoft Edge	127.548 ms	68.346 ms
Google Chrome	88.896 ms	52.587 ms
Mozilla Firefox	75.656 ms	74.768 ms

Table 6.2: Table showing the average execution times of the remuxing step with the web worker for a 2160p video

By having a closer look at the graph, there seem to be spikes which are on different segments for Microsoft Edge, Google Chrome and Mozilla Firefox. An explanation for

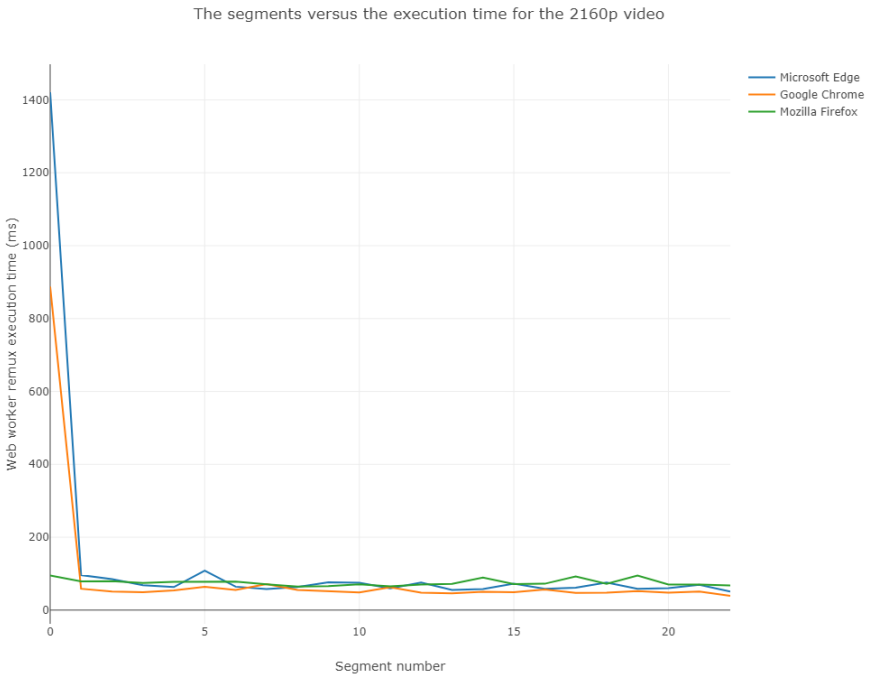


Figure 6.3: Graph showing the execution times of the web worker remuxing step per 1000 millisecond segment for a 2160p video

these spikes is yet to be found. The first assumption was the segment packages being bigger (or lower) in size for subsequent segments but then the spikes would be on the same segments which is not the case. If we look at segment 15 of Figure 6.4 we can see that this segment is a spike for Google Chrome but not for Microsoft Edge or Mozilla Firefox. Our assumption is proven invalid if we look at the segment sizes in Figure 6.5. Here we can see that the size of segment 15 is a lot smaller than the others. A possibility can be the Garbage collector kicking in and halting web worker execution in Google Chrome. We will analyze this more in Section 6.3.

6.2.3 1080p vs 2160p

By looking at the average execution times of the 1920 by 1080 pixels (1080p) and 3840 by 2160 pixels (2160p) video which are represented in Table 6.3, there are a few things to notice. When the implementation was started, the assumption was that the average execution times would differ as the packages that have to be read are approximately 4 times larger. This is because the 1080p video fits 4 times in the 2160p video. However, this is not represented in the numbers when looking at the execution times. Microsoft Edge web browser differs *only* by 1.935 milliseconds, Google Chrome by 0.479 milliseconds and Mozilla Firefox by 0.151 milliseconds. The differences were assumed because the memory copies that need to happen are larger for the 2160p video as more data has to be copied. Even though there is a small difference in execution time, there is not enough difference to say that a higher resolution video has an noteworthy impact on the processing time of the web worker.

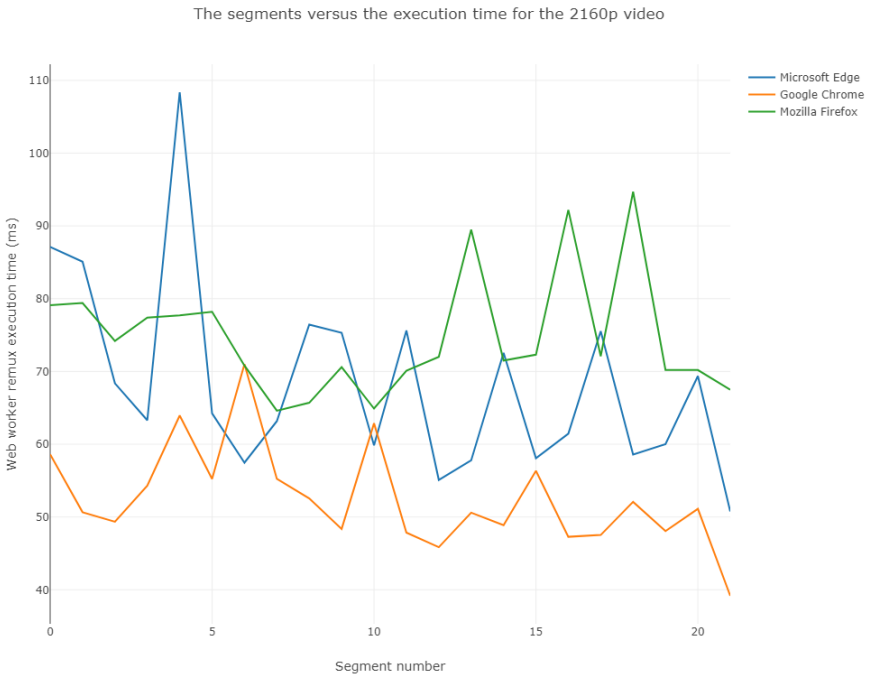


Figure 6.4: Graph showing the execution times of the web worker remuxing step per segment without the first segment for a 2160p video

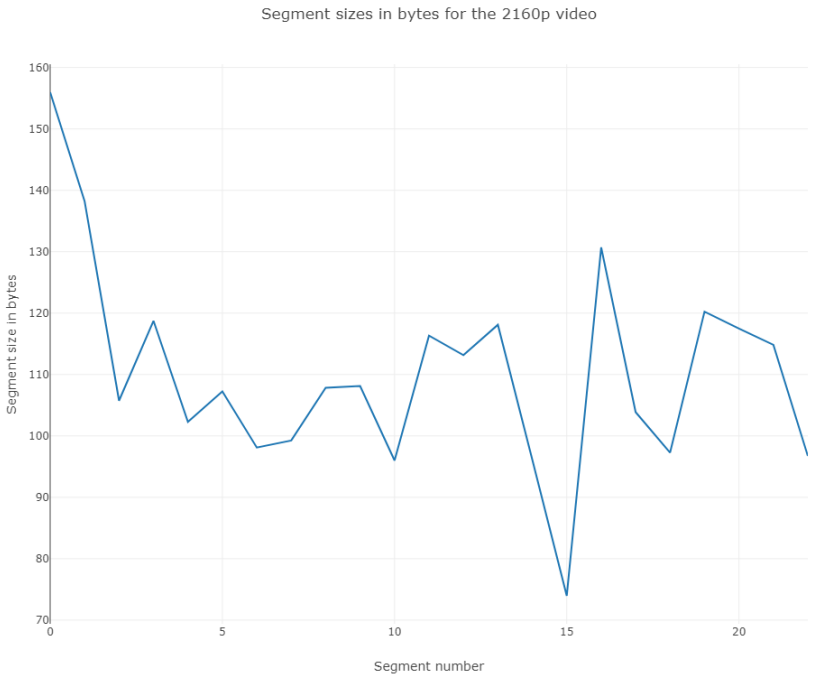


Figure 6.5: Graph showing the segment index correlating the input total size without the first segment for a 2160p 3x3 tiled video

	1080p	2160p
Microsoft Edge	66.411 ms	68.346 ms
Google Chrome	52.108ms	52.587 ms
Mozilla Firefox	74.617 ms	74.768 ms

Table 6.3: Comparison of the average execution time without first segment of the 1080p versus the 2160p video

6.2.4 1080p semgent duration 5000 milliseconds

Another test that has been done is the increase of segment durations to 5000 milliseconds instead of 1000 milliseconds for the 1080p video content. However, this is not a real life situation as a user wants the quality adaptation to happen as fast as possible. If a segment duration is 5000 milliseconds, a user would need to wait 5000 milliseconds (given that 1 segment is buffered and being the worst case that the segment just started) before the quality changes. As such, the duration of segments should always be as short as possible.

Just as in the other cases, the start up time is a lot higher than the subsequent executions which will be further analyzed in Section 6.3.

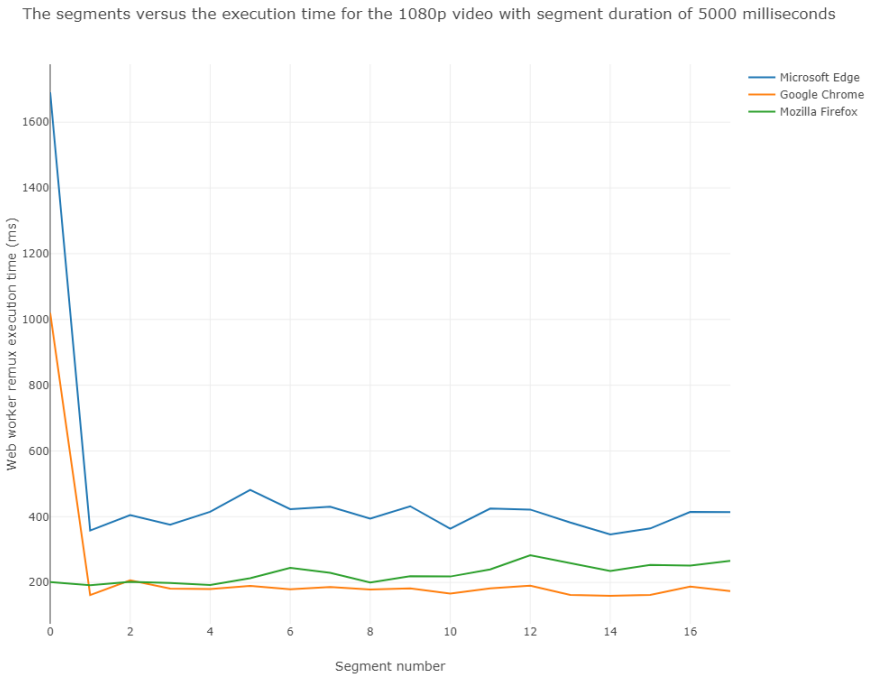


Figure 6.6: Graph showing the execution times of the web worker remuxing step per 5000 millisecond segment without the first segment for a 1080p video

In the graph of Figure 6.7, one thing stands out. The Microsoft Edge web browser has almost the double execution time than Google Chrome and Mozilla Firefox. As the segments are 5 times longer (from 1000 milliseconds to 5000 milliseconds) the assumption was that the average execution time would be 5 times as long. This assumption is based on the fact that the segment duration is 5 times longer and as such contains 5 times more video data. Hereby, needing to read 5 times as many headers and copy 5 times as much

data. However, the execution time is not 5 times as high for any of the execution times. The average execution time in Google Chrome is only 3.4 times higher, Mozilla Firefox 3 times and for Microsoft Edge 6 times higher. A reason for the extreme increase for Microsoft Edge and not for Mozilla Firefox nor Google Chrome was not found and needs further investigation in future work.

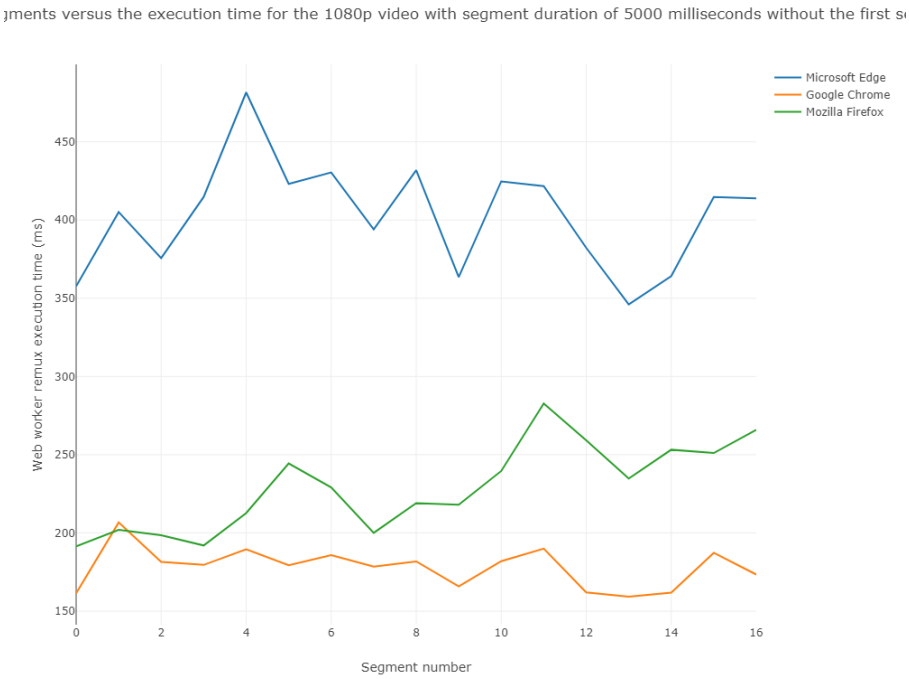


Figure 6.7: Graph showing the execution times of the web worker remuxing step per 5000 millisecond segment without the first segment for a 1080p video

	1080p	
	1000 ms	5000 ms
Microsoft Edge	66.411 ms	402.678 ms
Google Chrome	52.108ms	178.009 ms
Mozilla Firefox	74.617 ms	229.052 ms

Table 6.4: Comparison of the average execution time without first segment of the 1080p with 1000 milliseconds segments versus 5000 milliseconds

6.2.5 2160p 6x6 tiles

The last test in different sized files is done with a 3840x2160 pixel video with 6 by 6 tiles. This means that a tile will have a dimension of 640 by 360 pixels. These segments have a duration of 1000 milliseconds and will be compared against the web worker execution times of the 3x3 2160p test. With this test, the assumption was made that even though there are a lot more tiles, 36 vs 9. The execution time would be higher as there would be many small memory copies instead of a few larger ones. In table 6.5 the comparison is shown. Our first thought is in deed correct, the average execution times without the first segments are indeed higher for the 6x6 tiles. However, the executions times are higher,

the expectation difference was not expected to be this high; 12,953 ms for Microsoft Edge, 16,29 ms for Google Chrome and 8,696 ms for Mozilla Firefox. To understand why the executions are this much higher, a deeper analyses is needed in future work. One possibility is the time needed to copy the input files to the web worker, as many more files are needed to be copied (37 + the init file instead of 10 + init file). Another possibility is the concatenation step that takes longer in the execution (more details of the concatenation step in Chapter 5. As more files are being input, more files need to be concatenated before the input video bit stream can be read. This means that in the concatenation, more files need to be opened and copied to a new file which takes more time to correctly open and close the different files.

	2160p 3x3	2160p 6x6
Microsoft Edge	68.346 ms	81.299 ms
Google Chrome	52.587 ms	68.877 ms
Mozilla Firefox	74.617 ms	83.313 ms

Table 6.5: Comparison of the average execution time without first segment of the 2160p 3x3 tiled versus the 2160p 6x6 tiled video

6.3 Analyses

To analyze the high execution times, the developer tools of Google Chrome were used. It is important to notice that the web application takes a performance hit by having the developer tools open as logging information is processed.

In Figure 6.8, the first run for the 2160p output segment is shown. Our first assumption was that the *Just in time* compiler [Lau13] was the problem for the slow execution of the first segment. However, as can be seen in Figure 6.8 the compilation process happens before the execution. It is even so that the compilation happens before the web worker is started as the segments are not even downloaded. The segment downloads happen at the almost 1000 ms mark after which they are taken as input for the web worker and the web worker execution starts. So our assumption of why the first execution is so long in comparison with the others, is wrong and needs further examination. When looking at the figure, it also seems as if no other function is being executed for a long period of time. By looking at other executions, this same halt is noticed even though these halts are smaller in time.

A close up of another execution can be seen in Figure 6.9(in which the same halt but shorter, can be noticed). The first thing to observe is the complete execution time of the web worker which is 115.54 milliseconds. The execution of the *preRun* step is 3.01ms, this is the step that will mount the working directory of the web worker. After this is done, the *doRun* function is executed. This function is the converted code from the native application and executes for 28.66 milliseconds. Once the combination into one native tiled HEVC segment has been made and the execution is done, the data has to be copied to a location for the main thread to access the output. This is done in *postRun* and takes 0.71 milliseconds. With these timings, we can conclude that the execution of the web worker in which actual processing is done, is: 32,38ms. However, there is an execution gap of 83.16 milliseconds. When looking at this gap, there seem to happen no function calls till the almost end. This execution gap occurred in all the executions with the same

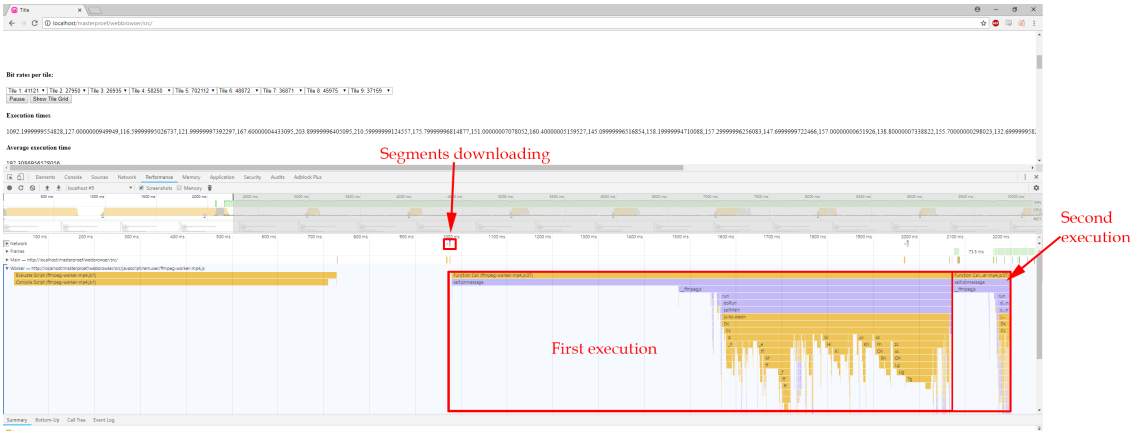


Figure 6.8: First run showing the JIT compiler first compiling the script before being executed

average time of 80 milliseconds for the video in 2160p. This gap (in smaller size) also occurred in the 1080p videos. An assumption could be made that this overhead is created by the web worker and is needed to initialize the web worker before execution could be done. However, this means that the overhead would be the same for all executions, regardless of 1080p or 2160p content which is not the case. Another assumption is that this overhead is generated by either the *Emscripten* conversion and extra *JavaScript* code is added which prevents immediate execution or ASM.JS itself. To be able to verify or reject this assumption, more research is needed.

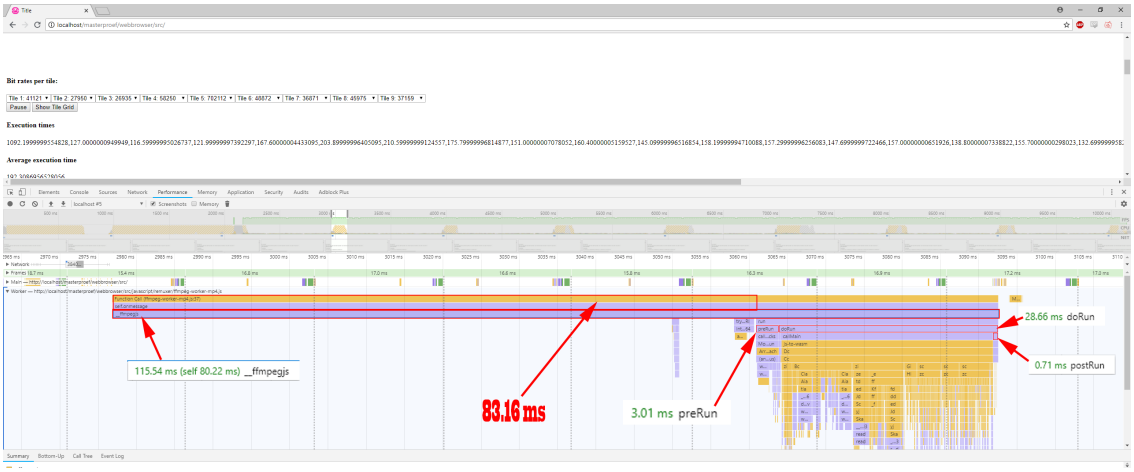


Figure 6.9: Analyses with the chrome developer tools

Chapter 7

Conclusion

This chapter will conclude this thesis by giving an overview of what has been done and discuss the future work possibilities. At the start of this thesis, we wondered if it was possible to stream tiled HEVC video over MPEG-DASH to a web browser, as Microsoft Edge at that time already supported HEVC decoding and Apple announced Safari for HEVC playback, to be released in their High Siera update. Furthermore, we were interested in the preprocessing speed to be able to play the HEVC video content in real-time in the web browsers.

The streaming of HEVC over MPEG-DASH works by abusing the tile feature. Tiles allow a video to be divided in rectangular independent regions. By using tiles and MPEG-DASH, the video content is split up so that every tile has its own **AdaptationSet** with **Representations**. One **AdaptationSet** is added containing all non video coding layer NAL units which are needed to be able to decode the whole video frame.

The first thesis question: *Is preprocessing a MPEG-DASH tiled video in a web browser to a native tiled HEVC video feasible?* has successfully been answered with yes. We implemented this by firstly creating a native application with *FFmpeg*. In this native application the different segments were preprocessed to eventually yield one DASH segment containing a natively tiled HEVC video. This native application was then transpiled to a web application. In the web application the quality of a tile changes based on the viewport and its position in regard to the tiles.

To answer our second research question: *Can the preprocessing be implemented in a sufficiently efficient manner to ensure live playback?* we measured the average preprocessing execution time in three popular contemporary web browsers. Throughout Chapter 6, these web browsers were tested with segments of a 1000 and 5000 milliseconds duration and video content in 1080p and 2160p with 3x3 and 6x6 tiles. Even though, there is yet to be an explanation for the high execution time of the first segment. We can conclude that live video playback is possible with the videos tested. However, more tests are needed to analyze the high start up times of the first segment processing.

Furthermore, we also tried if it was possible to decode the tiled HEVC video in the web browsers that lack native HEVC decoding support like Google Chome and Mozilla Firefox. This was done by software decoding the HEVC video content. The implementation worked in ASM.js which is a highly optimized subset of *JavaScript*. The code to execute the software decode was transpiled from the native application which was based on *FFmpeg*.

The ASM.JS script was then executed using a web worker. Our goal here was to be able to play the HEVC video content in any web browser. The software implementation in ASM.JS decoded a HEVC video frame to raw YUV and then encoded it to JPEG to be able to project the JPEG image in a HTML5 canvas element of the web browser. This did seem like a technically likely situation to work, even though it would be slow as we were decoding HEVC video in software. However, in practice it did not work because of the size of the YUV output per frame. The web browser could not handle the memory consumption and would eventually crash the web page or the web browser, making playback impossible.

7.1 Future Work

A first possibility for future work could be the optimization of the native implementation which is ported to a web browser application. In the current implementation as described in Phase 2 of Section 5.2 the input files are concatenated into one file. An optimization would be to use multiple input streams to read each file individually instead of first concatenating the files and then ready the constituting frames from this concatenated file. This optimization would decrease the processing time as no new file has to be created.

Secondly a smart bandwidth management system could be introduced. An example of such smart bandwidth management system is better distribution of tile qualities. It might be interesting to give the tiles surrounding the viewport a quality other than the lowest quality. This is especially helpful for situations were the viewport is bigger than a single tile. Another example is by looking at the user's viewport motion. Based on the movement, a tile quality decision could be made. For example, give beter qualities to the tiles in which direction the client is moving. Maybe even extent this by giving highest quality to the tiles in the viewport and downgrading the qualities in steps of the further the tiles are from the viewport in that specific direction.

In [FC16] there are some possible future improvements described. The difference in quality of tiles can be noticeable when distribution of bandwidth is done in a matter of giving max quality to the viewed tile and giving lowest to surrounding. While this might be a good approach in some situations. It could also show hard borders with many artifacts in other situations.

Chapter 8

Appendix A - Dutch Summary

H.264/MPEG-4 AVC is al jaren de standaard voor videoconsumptie. De huidige diversiteit aan services, de nog steeds groeiende populariteit van hoge kwaliteit video's en de voortdurend toenemende videoresolutie vragen om een efficiëntere codeermogelijkheid dan H.264/MPEG-4 AVC kan bieden [SO10] [SOHW12]. Bovendien genereren mobiele apparaten steeds meer downloadverkeer via draadloze netwerken die meer vatbaar zijn voor transportfouten en minder bandbreedte hebben dan hun bekabeld netwerk tegenhangers. Deze mobiele netwerken zijn op geen enkele manier geoptimaliseerd voor het streamen van videomateriaal in de gevraagde hoge kwaliteit. Hoewel de bandbreedtehoeveelheid van mobiele netwerken elke dag toenemen, is de behoefte aan betere videocompressie voor video's met hoge resolutie groter dan ooit om de steeds toenemende vraag naar video-inhoud via internet te voeden [SCF+12]. Bijgevolg is H.265/High Efficiency Video Coding (HEVC) ontworpen in een gezamenlijke inspanning van ITU-T VCEG en ISO/IEC MPEG-standaardisatieorganisaties om de tekortkomingen van H.264/AVC te verbeteren door hogere resoluties te ondersteunen en de hoeveelheid bitrate die gecodeerd video's vragen te verlagen. Met behulp van HEVC kan de grootte beperkt worden tot wel 50% voor video's in dezelfde kwaliteit [CAMJ+12].

Doordat HEVC een gevolg is van H.264 zijn er veel eigenschappen van H.264 overgenomen maar ook verbeterd. Een van de nieuwe kenmerken die standaard ondersteund wordt in HEVC is *tiles*. *Tiles* ondersteuning was aanwezig in H.264 via omwegen. Met behulp van *tiles*, kan een video in gelijkwaardige rechthoeken opgedeeld worden. Deze *tiles* kunnen dan elke afzonderlijk geëncodeerd en gedecodeerd worden. Het is ook mogelijk om de kwaliteit van een individuele of meerdere *tiles* te veranderen in dezelfde video. Dit door een of enkele *tiles* onderling te verwisselen met *tiles* in verschillende kwaliteiten.

De video waarmee gewerkt zal worden is een 360 graden video. In deze video zal de gebruiker de mogelijkheid hebben om rond te kijken. Dit door een gebruiker toe te staan in de 360 graden video rond te kijken met behulp van een *viewport*. De *viewport* is de regio van de video die bekeken wordt door de gebruiker. Onze doelstelling zal zijn om de video buiten het *viewport* een lage kwaliteit te geven en de videobeelden in het viewport een hoge kwaliteit.

Aangezien de video ook tot bij de gebruiker moet geraken over het Internet om zo de video in de web browser af te spelen, maken we gebruik van de MPEG-DASH media streaming standaard. MPEG-DASH stelt de gebruiker in staat om de kwaliteit van de video tijdens het afspelen te veranderen door videosegmenten van verschillende kwaliteiten uit te wisselen. Deze segmenten kunnen een zelfgekozen tijdspanne hebben. Aangezien onze implementatie door middel van *tiles* werkt, zal in de plaats van de volledige video, de verschillende *tiles* in een de mogelijke kwaliteiten gestreamd worden. Wanneer alle *tiles* in de hoogste kwaliteit doorgestuurd worden, zal de bandbreedte die nodig is om de video af te spelen enorm zijn. Zoals we in de inleiding vermeld hebben is het net de bedoeling om een zo laag mogelijke bandbreedte te halen zodat het doorsturen van de video efficiënt kan gebeuren. Dit wilt zeggen dat de *tile* die de grootste oppervlakte van het *viewport* inneemt in de beste kwaliteit gestreamd zal worden. Alle andere *tiles* zullen in onze opstelling de laagste kwaliteit krijgen.

Het uiteindelijke doel is om de 360 graden video in de web browser af te spelen. We kunnen alle *tiles* apart opvragen in de kwaliteit gebaseerd op het *viewport*. Het enige dat resteert is de video samenvoegen zodat deze afgespeeld kan worden in de web browser. Dit wordt gedaan aan de hand van een *web worker*. Een *web worker* is een stukje JavaScript code dat parallel uitgevoerd zal worden. Met parallel bedoelen we dat dit stukje JavaScript

code uitgevoerd kan worden zonder impact te hebben op de normale functionaliteiten van de web browser. In de *web worker* worden de verschillende *tiles* samengevoegd tot een enkele native *tiled* HEVC video. Deze video zal als input dienen voor het HTML5 video element dewelke de video presenteert aan de gebruiker. In deze masterproef werd de implementatie gemaakt die het toelaat om de *tiles* samen te voegen tot een native *tiled* HEVC video die gedecodeerd kan worden door de web browser. Verder werd in deze masterproef ook de uitvoeringstijden voor het verwerken van 1080p, 2160p, verschillende tijdspanne van de videosegmenten en veranderingen in het aantal *tiles* getest.

8.1 HEVC

H.265/HEVC is ontworpen in een gezamenlijke inspanning van ITU-T VCEG en ISO/IEC MPEG-standaardisatieorganisaties om de tekortkomingen van H.264/AVC te verbeteren door hogere resoluties te ondersteunen en door de bitrate vereisten van gecodeerde video's te verlagen. Dit wordt gedaan door een bestandsreductie tot 50% te realiseren met behoud van dezelfde kwaliteit [CAMJ+12].

HEVC ondersteunt verschillende benaderingen om de decodering en codering te parallelliseren. Een van deze benaderingen is het gebruik van *tiles*. Met *tiles* kan een gebruiker een video splitsen in rechthoekige gebieden die afzonderlijk kunnen worden gecodeerd en gedecodeerd.

Een ander belangrijk onderdeel van H.265/HEVC zijn de NAL-units. NAL unit staat voor *Network Abstraction Layer* en encapsuleren de geëncodeerde video data zodat deze opgeslagen kunnen worden [Wie14]. Er zijn twee soorten NAL units: *Video Coding Layer* en niet-*Video Coding Layer* NAL units. De VCL NAL unit bestaat uit alle *CTU* data en *slices*. *Slices* werden origineel in H.264/AVC ontwikkeld om een video bit stream in kleinere delen op te splitsen. Deze kleinere delen kunnen via het Internet verzonden worden. Later werden *slices* ook uitgebuit om parallellisatie mogelijk te maken in H.264. *CTU* staat voor *Coding Tree Unit* maar zal in deze sectie niet verder toegelicht worden. De uitleg *CTU* kan gelezen worden in de volledige thesis tekst. Het is echter wel belangrijk te weten dat de *CTU* voor de opdeling van het video frame zorgt. Deze *CTU*s zijn blokjes van 16x16, 32x32 of 64x64 pixels. *CTU*s worden gebruikt om de video beeld per beeld te reconstrueren.

8.2 MPEG-DASH

MPEG-DASH is een streaming standaard die ontwikkeld is door de toenemende groei van HTTP servers en de nood aan streaming zonder overhead aan serverkant zoals bij Real-Time Transport Protocol (RTP) wel het geval is. MPEG-DASH zal in deze masterproef ook gebruikt worden als de streaming methode.

MPEG-DASH werkt door middel van een Media Presentation Description (MPD) bestand. Dit bestand zal geparset moeten worden om de streaming sessie te starten. Een MPD omvat een aaneenschakeling van niet overlappende **Periods**, representaties van tijdsintervallen. Inhoudelijk bevatten **Periods** de verschillende coderingsparameters en serverlocaties van media componenten zoals: video's met verschillende codec, video's met

verschillende kijkhoeken, ondertitels of audio voor verschillende talen, audio die meer informatie bevat bijvoorbeeld regisseur opmerkingen, enzovoort [Sto11].

Een **Period** bestaat uit één of meerdere **AdaptationSets**. Elk van deze **AdaptationSets** is een high level representatie van een media component. Zo staat in dezelfde **AdaptationSet** een collectie van onderling verwisselbare geëncodeerde versies van een media component. In deze thesis zal elke **AdaptationSet** één *tile* voorstellen, behalve de eerste **AdaptationSet** die enkel de niet-VCL NAL units bevat zodat het totaal beeld gereconstrueerd kan worden. De NAL-units worden in Sectie 8.1 meer toegelicht.

Een **AdaptationSet** bestaat uit een set van **Representations**. De **Representations** stellen de kwaliteit van een media component voor. Zo kunnen twee **Representations** binnen dezelfde **AdaptationSet** eenzelfde video (in het geval van deze thesis, *tile*) voorstellen maar dan met een verschillende bitrate. Deze **Representation** kan tijdens het streamen van de video gewisseld worden.

8.2.1 Spatial Relationship Description

Om het streamen van *tiles* mogelijk te maken, is een uitbreiding nodig op de basisstructuur van de MPD, Spatial Relationship Description (SRD). Deze nieuwe extensie is geleverd met de tweede wijziging van MPD in ISO/IEC 23009-1:2014/Amd.2:2015 [iso15]. Via de SRD informatie, weet de DASH-*client* welke video *tiles* moeten worden aangevraagd op basis van de kijklocatie van de *client* in de video. Hierdoor kan een DASH-*client* ervoor kiezen om alle of slechts een deel van de betreffende video *tiles* te streamen. Dit laat ook toe om gebruikers bepaalde *tiles* van de video in hoge kwaliteit en andere in lage kwaliteit te streamen [DvdBTN16].

De SRD extensie voegt enkele nieuwe functies toe aan de MPD syntaxis. Een volledige beschrijving is te vinden in ISO/IEC 23009-1 [iso15]. Alleen de belangrijke onderwerpen in de context van deze masterscriptie zullen worden toegelicht. Het hoofdconcept is om een 2-dimensionale ruimte te definiëren voor de verschillende locaties van de media-objecten.

SRD introduceerde twee nieuwe tags: **EssentialProperty** en **SupplementalProperty**. Bovendien worden beide eigenschappen gedefinieerd in een **AdaptationSet**. Met de **EssentialProperty** kan de auteur van MPD bepalen dat het essentieel is om deze descriptor met te verwerken om te zorgen voor een correcte verwerking van de bovenliggende inhoud, de **AdaptationSet**. Wanneer oudere DASH-*client*s deze descriptor tegenkomen, negeren ze deze **AdaptationSet** [NTD+16]. Met de **SupplementalProperty** kan de auteur van het MPD bestand definiëren dat het niet essentieel is om de informatie van de descriptor correct te ontleden om te zorgen voor een correcte verwerking van de **AdaptationSet**.

Deze tags definiëren twee kenmerken: *schemeIdUri* en de *value* [NTD+16]. De *schemeIdUri* definieert het schema bij het lezen van waarden. In deze context is de waarde van *schemeIdUri* `urn:mpeg:dash:srd:2014`. De tag *value* is iets ingewikkelder gestructureerd en bevat de volgende velden in functie van de masterthesis:

- *source_id* is een verplicht veld dat bestaat uit een geheel getal dat de inhoudsbron identificeert. Met deze ID kan een *tile* naar een video verwijzen.

- *object_x* een niet-negatief geheel getal en vereist veld dat de horizontale positie vertegenwoordigt, beginnend in de linkerbovenhoek van het overeenkomstige media-element.
- *object_y* een niet-negatief geheel getal en vereist veld dat de verticale positie vertegenwoordigt, beginnend in de linkerbovenhoek van het overeenkomstige media-element.
- *object_width* is een verplicht veld dat de breedte van het overeenkomstige media-element met een niet-negatief geheel getal vertegenwoordigt.
- *object_height* is een verplicht veld dat de hoogte van het bijbehorende media-element met een niet-negatief geheel getal vertegenwoordigt.

8.3 Implementatie

De implementatie begon met een eenvoudig idee; *Stream tiled HEVC video over MPEG-DASH om 360 graden video te projecteren, zodat één decoder nodig is en adaptieve kwaliteitscontrole mogelijk is.* Vanaf het begin was bekend dat *GPAC* werk had geleverd rondom *tiled* HEVC adaptieve MPEG-DASH video streaming. Ze hebben deze implementatie ook gedemonstreerd op wetenschappelijke conferenties. Het enige dat aan het begin van het proefschrift gevonden werd, waren tegenstrijdige handleidingen over het genereren van *tiled* HEVC MPEG-DASH-inhoud, [Feu17] en [Feu18]. Deze handleiding is herschreven en is te vinden in de masterthesis.

Er is besloten om met een native toepassing te starten omdat de ondersteuning voor HEVC in de context van de webbrowser zo minimaal was dat alleen Edge op Windows HEVC ondersteunde. Apple had de ondersteuning van HEVC aangekondigd in zijn jaarlijkse keynote in 2017 [Inc18]. De release kwam in september met de nieuwe grote Mac OS X-update: 10.13 High Sierra. In beide webbrowsers was er een beperking om *tiled* video's niet toe te staan in afzonderlijke streams. Daarom werd de keuze gemaakt om een web toepassing te maken die van MPEG-DASH segmenten (waarin elke *tiles* zijn eigen **AdaptationSet** bevat en een **AdaptationSet** voor de niet VCL NAL units) van een *tiled* HEVC bit stream naar een enkele HEVC bit stream gaat. Deze bit stream kan de webbrowser decoderen en gebruik maken van hardware decodering.

8.3.1 Native applicatie

Er werd gekozen om eerst te starten met een native applicatie zodat de volledige HEVC bit stream begrepen werd. De ontwikkeling gebeurde in enkele delen en is begonnen met de native applicatie in *FFmpeg* versie 3.4.0.

Bij de eerste implementatie is er gestart met een MP4-bestand dat dezelfde structuur had als de MPEG-DASH stream. Een overzicht van de implementatie is te zien in Figuur 8.1. In het input MP4-bestand was er een stream per *tile* en een extra stream voor de niet VCL NAL units. Deze extra stream bevatte ook de complete informatie van de video. Om een enkele samengevoegde video bit stream te vormen, werd van elke stream een pakket gelezen. Zo'n pakket bevatte voor de eerste stream de niet VCL NAL units, voor de andere streams was dit een geëncodeerde *tile*. Wanneer van elke stream een pakket gelezen was,

werden deze pakketten in de volgorde van de *tiles* achter elkaar geplaatst in een nieuw pakket. Dit pakket werd dan op zijn beurt uitgeschreven naar een MP4-container.

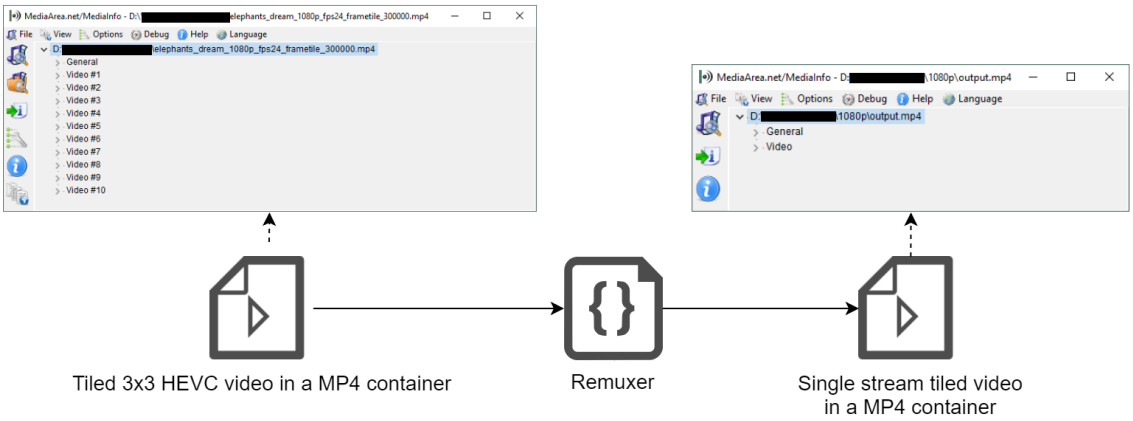


Figure 8.1: Eerste fase van een MP4 container met een tile per stream naar een enkele stream met tiles

In het volgende deel vonden twee stappen tegelijk plaats. Afbeelding 8.2 is hier een weergave van. De uiteindelijke output is een enkel segment dat een native *tiled* HEVC bit stream bevat. Verder zal ook een *init.mp4* geproduceerd worden, hierover later meer in Sectie 8.3.2. Om tot de deze stap te komen wordt de input samengevoegd tot een enkel bestand. De input bevat een *init.mp4*, dit bestand bevat net zoals het MP4-bestand uit de vorige stap alle informatie die nodig is over de verschillende *tiles*. De verschillende segmenten bevatten de verschillende stream data. Wanneer deze bestanden samengevoegd zijn, is het mogelijk om dit bestand op exact dezelfde manier als in de vorige stap te lezen. Van elke stream zal er een packet gelezen en weggeschreven worden naar de output die nu een segment is in plaats van een MP4 bestand.

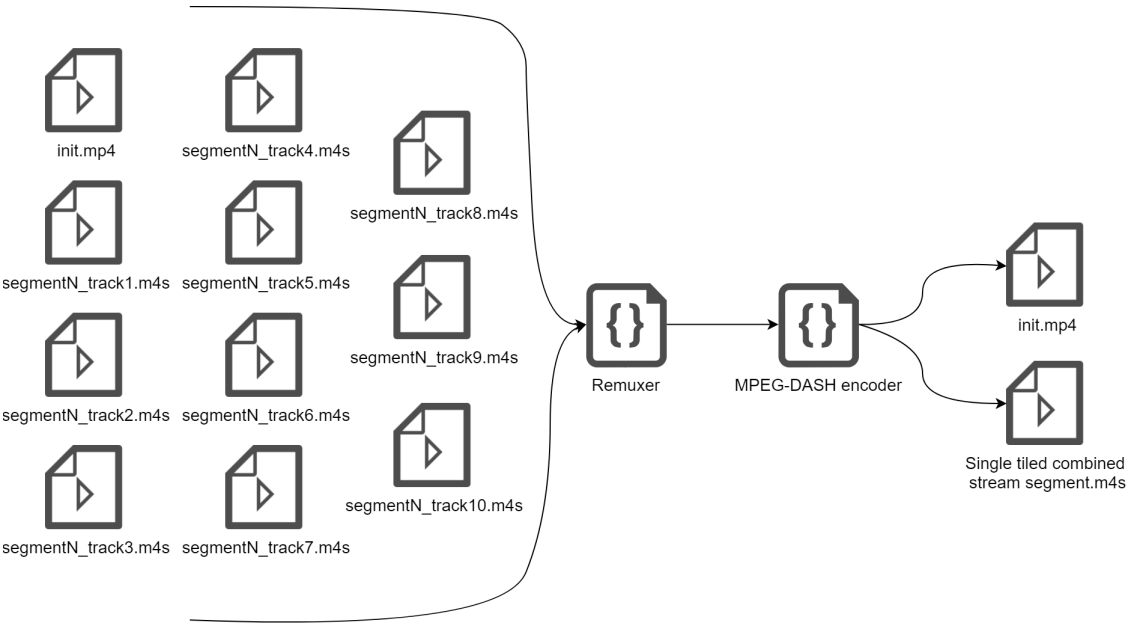


Figure 8.2: Fase 2 program van segmenten met een stream per tile HEVC video (en een niet VCL NAL-unit stream) naar een enkele native tiled HEVC MPEG-DASH stream

Als laatste native applicatie hebben we getest of het software matig decoderen van de HEVC video naar raw YUV of JPEG per video afbeelding mogelijk was. Dit is bij de native applicatie gelukt, maar is echter niet geïmplementeerd in de web browser door de grootte hoeveelheid geheugen en rekenkracht die nodig is.

8.3.2 Web applicatie

De web implementatie gaat via een *web worker* werken voor de MPEG-DASH segmenten te verwerken. Een *web worker* is een nieuwe thread die asynchroon *JavaScript* code kan uitvoeren. Door het omzetten van de implementatie van de tweede fase naar ASM.JS is het mogelijk deze op de *web worker* uit te voeren. ASM.JS is een zeer strikte subset van *JavaScript* en heeft veel C-kenmerken [Bam18]. Het is niet nodig om de details van ASM.JS te begrijpen voor deze masterscriptie, maar het is belangrijk om te begrijpen dat de uiteindelijke conversie van de C-implementatie ASM.JS code oplevert dewelke zeer geoptimaliseerde *JavaScript* is.

De *scheduler* is een algoritme dat de desbetreffende MPEG-DASH segmenten zal opvragen. Afhankelijk van het algoritme dat gebruikt wordt, worden segmenten (*tiles*) in specifieke kwaliteiten opgevraagd. In het geval van deze masterthesis zal dit zijn op basis van het *viewport*. Met behulp van het *viewport* wordt er gekeken naar de *tiles* die deze bevat. De *tile* die de meeste oppervlakte van het *viewport* inneemt zal de hoogste kwaliteit krijgen. Wanneer alle segmenten voor een enkel segment te vormen opgevraagd zijn, kan het omzetten via de *web worker* beginnen. Eens het omzetten gedaan is, wordt er gekeken of het video element geïnitialiseerd moet worden. Het video element werkt op basis van een *media source* buffer. Deze buffer zal alle video data bevatten om dan de video te decoderen. Voor het toevoegen van het eerste segment moet de initialisatie door middel van het *init.mp4* bestand gebeuren. Eens de *media source* buffer genoeg data bevat, kan het decoderen en afspelen van de video beginnen.

8.4 Conclusie

Aan het begin van de masterproef vroegen we ons af of het mogelijk was om *tiled* HEVC-video over MPEG-DASH naar een webbrowser te streamen terwijl Microsoft Edge HEVC ondersteund en Apple Safari de webbrowser-ondersteuning voor HEVC aankondigde. Verder waren we geïnteresseerd in de voorbewerkingssnelheid om de HEVC video-inhoud in de webbrowser te kunnen afspelen.

Het streamen van HEVC over MPEG-DASH werkt door misbruik te maken van de *tiles*. Met behulp van *tiles* kan een video verdeeld worden in rechthoekige onafhankelijke regio's. Door *tiles* en MPEG-DASH te gebruiken, wordt de video-content opgesplitst, zodat elke *tile* zijn eigen segmenten in verschillende kwaliteiten heeft. Dit wilt zeggen in MPD termen: één **AdaptationSet** per *tile* en één **AdaptationSet** met alle niet-videocoderingslagen NAL-eenheden die nodig zijn om het hele videoframe te kunnen decoderen.

De eerste scriptievraag: *Is het voorbewerken van een MPEG-DASH tiled video in een webbrowser naar een native tiled HEVC video mogelijk?* is succesvol beantwoord met ja. We hebben dit geïmplementeerd door eerst een native applicatie te maken met *FFmpeg*. In deze native toepassing worden de verschillende segmenten voorbewerkt tot uiteindelijk

één DASH-segment met een native *tiled* HEVC video. Deze native applicatie werd vervolgens getranscodeerd naar een webapplicatie. In de web toepassing kan de gebruiker de *tile* kwaliteiten wijzigen op basis van het *viewport* in de 360 graden video. De tweede vraag waarin we vroegen: *Hoe efficiënt is het voorbewerken om live afspelen te garanderen*, kan worden beantwoord met een gemiddelde uitvoeringstijd per webbrowser. We kunnen ook bevestigen dat de voorbewerkingssnelheid laag genoeg is om continu afspelen te garanderen. Dit werd getest met segmenten met een duur van 1000 en 5000 milliseconden. Ook is er getest met 1080p en 2160p video's. Daarnaast is het aantal *tiles* voor de 2160p video getest met een 3x3 en 6x6 structuur.

Verder hebben we ook geprobeerd of het mogelijk was om de *tiled* HEVC video in de webbrowser te decoderen. Ons doel hierbij was om de HEVC video inhoud in elke webbrowser te kunnen afspelen. Dit is gedaan door het decoderen van een HEVC videoframe naar een onbewerkte YUV frame en vervolgens te encoderen naar JPEG om de JPEG-afbeelding in een canvaselement van de webbrowser te kunnen projecteren. Dit leek echter een theoretisch waarschijnlijke situatie om te werken. We hadden echter wel de verwachting dat dit traag zou zijn omdat we de video in software decoderen. In de praktijk werkte het echter niet vanwege de grootte van de YUV bestanden per decodering stap. De webbrowser kon het decoderen niet aan en uiteindelijk zou de webpagina of de webbrowser crashen, waardoor afspelen onmogelijk werd.

Bibliography

[Ado] Adobe. Adobe HTTP Dynamic Streaming (HDS). `http://www.adobe.com/devnet/hds.html`. [Online; accessed 02-07-2018].

[Arc18] Jake Archibald. JavaScript Promises: an Introduction. `https://developers.google.com/web/fundamentals/primers/promises`, 2018. [Online; accessed 14-08-2018].

[Bam18] Will Bamberg. What is asm.js, exactly? `https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js`, 2018. [Online; accessed 14-08-2018].

[CAMJ+12] Chi Ching Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl. Parallel scalability and efficiency of HEVC parallelization approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1827–1838, dec 2012. `doi:10.1109/tcsvt.2012.2223056`.

[CFD+17] Cyril Concolato, Jean Le Feuvre, Franck Denoual, Eric Nassor, Nael Ouedraogo, and Jonathan Taquet. Adaptive streaming of HEVC tiled videos using MPEG-DASH. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 1–1, 2017. `doi:10.1109/tcsvt.2017.2688491`.

[DSA+11] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362, oct 2011. `doi:10.1145/2043164.2018478`.

[DvdBTN16] Lucia D'Acunto, Jorrit van den Berg, Emmanuel Thomas, and Omar Niamut. Using MPEG DASH SRD for zoomable and navigable video. In *Proceedings of the 7th International Conference on Multimedia Systems - MMSys '16*. ACM Press, 2016. `doi:10.1145/2910017.2910634`.

[ems18] Emscripten. `http://emscripten.org`, 2018. [Online; accessed 14-08-2018].

[Eng] Ralf S. Engelschall. ECMAScript 6 — New Features: Overview & Comparison. `http://es6-features.org/`. [Online; accessed 15-07-2018].

[FC16] Jean Le Feuvre and Cyril Concolato. Tiled-based adaptive streaming using MPEG-DASH. In *Proceedings of the 7th International Conference on Multimedia Systems - MMSys '16*. ACM Press, 2016. `doi:10.1145/2910017.2910641`.

[Feu17] Jean Le Feuvre. HEVC Tile-based adaptation guide. https://gpac.wp.imt.fr/2017/02/01/hevc-tile-based-adaptation-guide/, 2017. [Online; accessed 24-07-2018].

[Feu18] Jean Le Feuvre. Tiled Streaming. https://github.com/gpac/gpac/wiki/Tiled-Streaming, 2018. [Online; accessed 24-07-2018].

[FFm18] FFmpeg. FFmpeg. https://github.com/FFmpeg/FFmpeg, 2018. [Online; accessed 09-08-2018].

[GNA14] Praveen GB, Prashanth NS, and Ramakrishna Adireddy. Analysis of HEVC/H265 Parallel Coding Tools. http://pathpartner.wpengine.com/wp-content/uploads/2016/09/PathPartner_WhitePaper_Analysis-Of-HEVC-Parallel-Tools-1.pdf, 2014. [Online; accessed 03-08-2018].

[GPA18] GPAC. GPAC a multimedia framework. https://github.com/gpac/gpac, 2018. [Online; accessed 09-08-2018].

[GR08] Sacha Goedegebure and Ton Roosendaal. Big Buck Bunny. https://peach.blender.org/, 2008.

[GTM17] Mario Graf, Christian Timmerer, and Christopher Mueller. Towards bandwidth efficient adaptive streaming of omnidirectional video over HTTP. In *Proceedings of the 8th ACM on Multimedia Systems Conference - MMSys'17*. ACM Press, 2017. doi:10.1145/3083187.3084016.

[HS16] Mohammad Hosseini and Viswanathan Swaminathan. Adaptive 360 VR video streaming: Divide and conquer! *CoRR*, abs/1609.08729, 2016. URL: http://arxiv.org/abs/1609.08729, arXiv:1609.08729.

[HS17] Mohammad Hosseini and Viswanathan Swaminathan. Adaptive 360 VR video streaming based on MPEG-DASH SRD. *CoRR*, abs/1701.06509, 2017. URL: http://arxiv.org/abs/1701.06509, arXiv:1701.06509.

[Inc18] Apple Inc. Advances in HTTP Live Streaming. https://developer.apple.com/videos/play/wwdc2017/504, 2018. [Online; accessed 09-08-2018].

[ISO14a] Information technology – coding of audio-visual objects – part 10: Advanced video coding. Standard, International Organization for Standardization, 2014. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:14496:-10:ed-8:v1:en.

[ISO14b] Iso/iec 23009-1:2014 - information technology – dynamic adaptive streaming over http (dash) – part 1: Media presentation description and segment formats. Standard, International Organization for Standardization, 2014. URL: https://www.iso.org/standard/65274.html.

[iso15] Iso/iec 23009-1:2014/amd 2:2015 - spatial relationship description, generalized url parameters and other extensions. Standard, International Organization for Standardization, 2015. URL: https://www.iso.org/standard/66486.html.

[ISO17] Iso/iec dis 23008-2: Information technology – high efficiency coding and media delivery in heterogeneous environments – part 2: High efficiency

video coding. Standard, International Organization for Standardization, 2017. URL: https://www.iso.org/standard/69668.html.

[IT18] ITU-T. H.265 : High efficiency video coding. http://handle.itu.int/11.1002/1000/13433, 2018.

[Kag18] Kagami. ffmpeg.js. https://github.com/Kagami/ffmpeg.js, 2018. [Online; accessed 09-08-2018].

[KML+12] Il-Koo Kim, Junghye Min, Tammy Lee, Woo-Jin Han, and JeongHoon Park. Block partitioning structure in the HEVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1697–1706, dec 2012. doi:10.1109/tcsvt.2012.2223011.

[Kuz16] Evgeny Kuzyakov. Next-generation video encoding techniques for 360 video and VR. https://code.fb.com/virtual-reality/next-generation-video-encoding-techniques-for-360-video-and-vr/, 2016. [Online; accessed 19-08-2018].

[kva18] Kvazaar. https://github.com/ultravideo/kvazaar, 2018. [Online; accessed 14-08-2018].

[Lau13] Thibault Laurens. How the V8 engine works? http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/, 2013. [Online; accessed 23-08-2018].

[Lon15] Brendan Long. The structure of an MPEG-DASH MPD. https://www.brendanlong.com/the-structure-of-an-mpeg-dash-mpd.html, 2015. [Online; accessed 09-07-2018].

[LSNHS05] Liang Lu, Rei Safavi-Naini, Jeffrey Horton, and Willy Susilo. On securing RTP-based streaming content with firewalls. In *Cryptology and Network Security*, pages 304–319. Springer Berlin Heidelberg, 2005. doi:10.1007/11599371_25.

[MAP+10] Aditya Mavlankar, Piyush Agrawal, Derek Pang, Sherif Halawa, Ngai-Man Cheung, and Bernd Girod. An interactive region-of-interest video streaming system for online lecture viewing. In *2010 18th International Packet Video Workshop*. IEEE, dec 2010. doi:10.1109/pv.2010.5706821.

[Mic18] Microsoft. About YUV Video. https://docs.microsoft.com/en-us/windows/desktop/medfound/about-yuv-video, 2018. [Online; accessed 07-08-2018].

[Mot12] Ito Motonari. HEVC – What are CTU, CU, CTB, CB, PB, and TB? https://codesequoia.wordpress.com/2012/10/28/hevc-ctu-cu-ctb-cb-pb-and-tb/, 2012. [Online; accessed 29-07-2018].

[MSH+13] Kiran Misra, Andrew Segall, Michael Horowitz, Shilin Xu, Arild Fuldseth, and Minhua Zhou. An overview of tiles in HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):969–977, dec 2013. doi:10.1109/jstsp.2013.2271451.

[MSW03] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/AVC video compression standard. *IEEE*

Transactions on Circuits and Systems for Video Technology, 13(7):620–636, jul 2003. doi:10.1109/tcsvt.2003.815173.

[Mue15a] Christopher Mueller. Apple HTTP Live Streaming. https://bitmovin.com/apple-http-live-streaming-hls/, 2015. [Online; accessed 02-07-2018].

[Mue15b] Christopher Mueller. Microsoft Smooth Streaming. https://bitmovin.com/microsoft-smooth-streaming/, 2015. [Online; accessed 02-07-2018].

[Mue15c] Christopher Mueller. MPEG-DASH (Dynamic Adaptive Streaming over HTTP, ISO/IEC 23009-1). https://bitmovin.com/mpeg-dash/, 2015. [Online; accessed 02-07-2018].

[Mul18] Solveig Multimedia. Zond 265 - HEVC Video Analyzer. http://www.solveigmm.com/en/products/zond/, 2018. [Online; accessed 01-08-2018].

[NTD+16] Omar A. Niamut, Emmanuel Thomas, Lucia D'Acunto, Cyril Concolato, Franck Denoual, and Seong Yong Lim. MPEG DASH SRD. In *Proceedings of the 7th International Conference on Multimedia Systems - MMSys '16*. ACM Press, 2016. doi:10.1145/2910017.2910606.

[OAS17] Cagri Ozcinar, Ana De Abreu, and Aljosa Smolic. Viewport-aware adaptive 360° video streaming using tiles for virtual reality. *CoRR*, abs/1711.02386, 2017. URL: http://arxiv.org/abs/1711.02386, arXiv:1711.02386.

[Ple18] Plex. Remuxing Files to MKV. https://support.plex.tv/articles/201097958-remuxing-files-to-mkv/, 2018. [Online; accessed 09-08-2018].

[RCAFE+14] Damian Ruiz-Coll, Velibor Adzic, Gerardo Fernandez-Escribano, Hari Kalva, Jose Luis Martinez, and Pedro Cuenca. Fast partitioning algorithm for HEVC intra frame coding using machine learning. In *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE, oct 2014. doi:10.1109/icip.2014.7025835.

[SCF+12] Rickard Sjoberg, Ying Chen, Akira Fujibayashi, Miska M. Hannuksela, Jonatan Samuelsson, Thiow Keng Tan, Ye-Kui Wang, and Stephan Wenger. Overview of HEVC high-level syntax and reference picture management. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1858–1870, dec 2012. doi:10.1109/tcsvt.2012.2223052.

[SHWW12] Thomas Schierl, Miska M. Hannuksela, Ye-Kui Wang, and Stephan Wenger. System layer integration of high efficiency video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1871–1884, dec 2012. doi:10.1109/tcsvt.2012.2223054.

[SO10] Gary J. Sullivan and Jens-Rainer Ohm. Recent developments in standardization of high efficiency video coding (HEVC). In Andrew G. Tescher, editor, *Applications of Digital Image Processing XXXIII*. SPIE, aug 2010. doi:10.1117/12.863486.

[Sod11] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE Multimedia*, 18(4):62–67, apr 2011. doi:10.1109/mmul.2011.71.

[SOHW12] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, dec 2012. doi:10.1109/tcsvt.2012.2221191.

[SSHS16] Robert Skupin, Yago Sanchez, Cornelius Hellge, and Thomas Schierl. Tile based HEVC video for head mounted displays. In *2016 IEEE International Symposium on Multimedia (ISM)*. IEEE, dec 2016. doi:10.1109/ism.2016.0089.

[SSP+17] Robert Skupin, Yago Sanchez, Dimitri Podborski, Cornelius Hellge, and Thomas Schierl. HEVC tile based streaming to head mounted displays. In *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, jan 2017. doi:10.1109/ccnc.2017.7983191.

[SSS15] Y. Sanchez, R. Skupin, and T. Schierl. Compressed domain video processing for tile based panoramic streaming using HEVC. In *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE, sep 2015. doi:10.1109/icip.2015.7351200.

[Sto11] Thomas Stockhammer. Dynamic adaptive streaming over HTTP –. In *Proceedings of the second annual ACM conference on Multimedia systems - MMSys '11*. ACM Press, 2011. doi:10.1145/1943552.1943572.

[Sul] Gary J. Sullivan. High Efficiency Video Coding HEVC. http://what-when-how.com/Tutorial/topic-397pct9eq3/High-Efficiency-Video-Coding-HEVC-12.html. [Online; accessed 01-08-2018].

[SWH+05] Thomas Stockhammer, Magnus Westerlund, Miska M. Hannuksela, David Singer, and Stephan Wenger. RTP Payload Format for H.264 Video. RFC 3984, February 2005. URL: https://rfc-editor.org/rfc/rfc3984.txt, doi:10.17487/RFC3984.

[web18a] Using Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers, 2018. [Online; accessed 14-08-2018].

[web18b] WebAssembly. https://webassembly.org/, 2018. [Online; accessed 14-08-2018].

[Wie14] Mathias Wien. *High Efficiency Video Coding*. Springer-Verlag GmbH, 2014. URL: https://www.ebook.de/de/product/22561599/mathias_wien_high_efficiency_video_coding.html.

[WSS+16] Ye-Kui Wang, Yago Sanchez, Thomas Schierl, Stephan Wenger, and Miska M. Hannuksela. RTP Payload Format for High Efficiency Video Coding (HEVC). RFC 7798, March 2016. URL: https://rfc-editor.org/rfc/rfc7798.txt, doi:10.17487/RFC7798.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Viewport dependent MPEG-DASH streaming of 360 degree natively tiled HEVC video in web browser context

Richting: **master in de informatica**
Jaar: **2018**

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 overtreedt.

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,

Moesen, Gert

Datum: **23/08/2018**