

**ISO/IEC JTC 1/SC 29/WG 11**

**Coding of moving pictures and audio**

**Convenorship: UNI (Italy)**

**ISO/IEC JTC 1/SC 29/WG 11 N18795**

**Document type: Approved WG 11 document**

**Title:** **Test Model 3 for Immersive Video**

**Status: Output document**

**Date of document: 2019-11-08**

**Source: Video**

**Expected action:**

**No. of pages: 37**

**Email of convenor: leonardo@chiariglione.org**

**Committee URL: mpeg.chiariglione.org**

**Editors’ integration notes:**

TMIV3.0 – N18795:

* m49958: Intel response to CE1: Grouping.
* m49962: Philips response to CE2: Pruning.
* m50949: Object-based immersive coding.
* m51439: Depth occupancy coding.
* m51487: Viewing space.

TMIV2.0 – N18577:

* No tool adoption.
* Inclusion of operating modes.

TMIV1.0 – N18470:

* Document established based on CfP responses reviewed during MPEG 126th meeting.

**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 11 N18795**

**Geneva, CH – October 2019**

|  |  |
| --- | --- |
| **Source:** | Video |
| **Title:** | Test Model 3 for Immersive Video |
| **Editors:** | Basel Salahieh, Bart Kroon, Joel Jung, Marek Domański |

**Test Model 3 for Immersive Video**

**Abstract**

The Video sub-group established the third working draft and the third test model algorithm description during the 128th MPEG meeting (7 to 11 October 2019, Switzerland) after evaluating the core experiment results and related contributions. This document serves as a source of general tutorial information on the MPEG Immersive Video (MIV) design. It defines the terminology used, the process and data flow, the operating modes, the algorithmic components, and the conformance points (i.e. data formats) adopted by the video group for the reference Test Model for Immersive Video (TMIV) at both the encoder & the decoder sides along with the general characteristic of the metadata and bitstream.

# Introduction

The MPEG-I project (ISO/IEC 23090) on coded representation of immersive media supports 3 Degrees of Freedom (3DoF), where a user’s position is static but its head can yaw, pitch and roll. This is available under MPEG-I Part 2 Omnidirectional MediA Format (OMAF) version 1 published in 2018. However, rendering flat 360° video, i.e. supporting head rotations only, may generate visual discomfort especially when objects close to the viewer are rendered. 6DoF enables translation movements in horizontal, vertical, and depth directions in addition to 3DoF orientations. The translation support enables interactive motion parallax providing viewers with natural cues to their visual system and resulting in an enhanced perception of volume around them. At the 125th MPEG meeting, a call for proposals [1] was issued to enable head-scale movements within a limited space.

This document describes the third version of the Test Model for Immersive Video (TMIV) that was defined at the 128th meeting of MPEG in October 2019. It is aligned with the TMIV 3 reference software. It includes a description of the process flow, the operating modes, the algorithmic components, and the data formats.

# Scope

The normative decoding process for MPEG Immersive Video (MIV) is specified in the working draft (WD) specification MPEG/N18794 [2]. The TMIV reference software (TMIV-SW) is provided to demonstrate a reference implementation of non-normative encoding and rendering techniques and the normative decoding process for MIV standard. The software is available publicly on Gitlab as detailed in Section 6.

This document provides an algorithmic description for the encoder and decoder sides of TMIV reference software. The purpose of this document is to promote a common understanding of the coding features and the reference methods supported in the TMIV-SW, in order to facilitate the assessment of the technical impact of new technologies during the standardization process. Common test conditions are provided in MPEG/N18789 [3].

# General Description of the System and Algorithms

## Terminology definitions

For the purpose of this document, the following terms and definitions apply. The \* superscript next to the term indicates that same definition is shared with the MIV WD3 document [2].

Table : Terminology definitions used for TMIV**.**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| *3D scene\** | Visual content in the *global reference coordinate system.* |
| *Access unit\** | A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order* and contain at most one *coded picture* with any specific value of nuh\_layer\_id. [This definition is an exact copy from HEVC Annex F.] |
| *Aggregator* | An embodiment of a process that accumulates pruning masks over an intra-period to account for motion within the scene. |
| *Additional view* | A *view representation* produced by the *view optimizer* operating on *source views* and is pruned and packed in multiple patches*.* |
| *Atlas\** | Aggregation of *patches* from one or more *view representations* after a packing process, into a picture pair which contains a *texture component* *picture* and a corresponding *depth component* *picture*. |
| *Atlas component\** | A texture or depth component of an *atlas*. |
| *Atlas list\** | A list of one or more *atlases* which may be present within the same *access unit*. |
| *Atlas parameters list* | Define how *patches* are packed within the *atlas(es)* and mapped to specific *view representations* in addition to the patches’ size and rotation within the *atlas(es)*. |
| *Atlas patch occupancy map\** | A 2D array corresponding to an *atlas* whose values indicate for each sample position in the *atlas* which *patch* the sample corresponds to, or if the sample is invalid. |
| *Basic view* | A *view representation* produced by the *view optimizer* operating on *source views* and is packed as a whole in a single patch*.* |
| *Camera parameters\** | Defines the projection used to generate a *view representation* from a 3D scene, including intrinsic and extrinsic parameters. |
| *Camera parameters list\** | A list of one or more *camera parameters*. |
| *Inpainter* | An embodiment of a process to fill missing regions prior to outputting a requested *target view*. |
| *Entity* | An abstract concept to be defined in another standard. For example, entities may either represent different physical objects, or a segmentation of the scene based on aspects such as reflectance properties, or material definitions. |
| *Entity layers* | A multi-layer representation of a *view representation* where each layer has only regions that belong to a single *entity*. |
| *Entity maps* | Multi-level maps indicating pixel-wise the entity that the texture and depth component belong to. |
| *Entity separator* | An embodiment of a process to turn a *view representation* into multiple *entity layers* based on the associated *entity map.* |
| *IV access unit parameters* | A set of parameters that desribes the *access unit,* the *atlases,* and the *patches* including *atlas parameters list.* |
| *IV sequence parameters* | A set of parameters that descrips the sequence including *camera parameters list.* |
| *Metadata merger* | An embodiment of a process to merge the metadata from various encoders each operating on different group. |
| *Omnidirectional view* | A *view representation* that enables rendering according to the user's *viewing orientation*, if consumed with a head-mounted device, or according to user's desired *viewport*, otherwise, as if the user was in the spot where and when the view was captured. |
| *Patch\** | A rectangular region within an *atlas* that corresponds to a rectangular region within a *view representation*. |
| *Patch descriptor\** | A description of the *patch*, containing its size, location within an *atlas*, rotation within an *atlas*, and location within a *view representation*. |
| *Patch packer* | An embodiment of a process to gather *patches* in an *atlas*. |
| *Projection\** | Inverse of the process by which the sample values of a projected *texture component* picture of a *view representation* are mapped to a set of positions in a *3D scene* represented in the *global reference coordinate system* according to the corresponding *depth* sample value and *camera parameter*s *list*. |
| *Pruner* | An embodiment of a process to identify and extract the occluded regions across *basic views* and *additional views* resulting in *patches*. |
| *Renderer\** | An embodiment of a process to create a *viewport* or *omnidirectional view* from a *3D scene* representation corresponding to a viewing position and orientation. |
| *Source splitter* | An embodiment of a process to automatically distribute views into different groups that get encoded separately and merged into the bitstream. |
| *Source view* | Indicates *source* video material before encoding that corresponds to the format of a *view representation*, which may have been acquired by capture of a *3D scene* by a real camera or by *projection* by a virtual camera onto a surface using source *camera parameters*. |
| *Target view* | Indicates either *perspective viewport* or *omnidirectional view* at the desired viewing position and orientation. |
| *View optimizer* | An embodiment of a process in charge of selecting *basic* and *additional* *views.* |
| *View representation\** | 2D sample arrays of a *texture component* and a corresponding *depth component* representing the projection of a *3D scene* onto a surface using *camera parameters*. |
| *Viewing space\** | Domain constraints for a good *viewport* rendering. The domain is defined in the 3D global space and also related to the viewing direction. It defines a value between 0 and 1 for every point in space for a given direction of the viewport, to be used by the application typically for fading. |
| *Viewport\** | Projection of texture onto a planar surface of a field of view of an omnidirectional or 3D image or video suitable for display and viewing by the user with a particular viewing position and orientation. |

## Process and data flow

A high-level diagram of the TMIV group-based encoder is presented in Figure 1. In essence, the TMIV encoding stage at a higher level consists of a source splitter that splits the views in multiple groups, multiplicity of TMIV encoders each encoding one of the groups, and metadata merger merging the resulting metadata from all groups operations. The per-group encoders operate independent of each other. Texture and depth is encoded using a generic 2D video encoder such as HEVC codec.

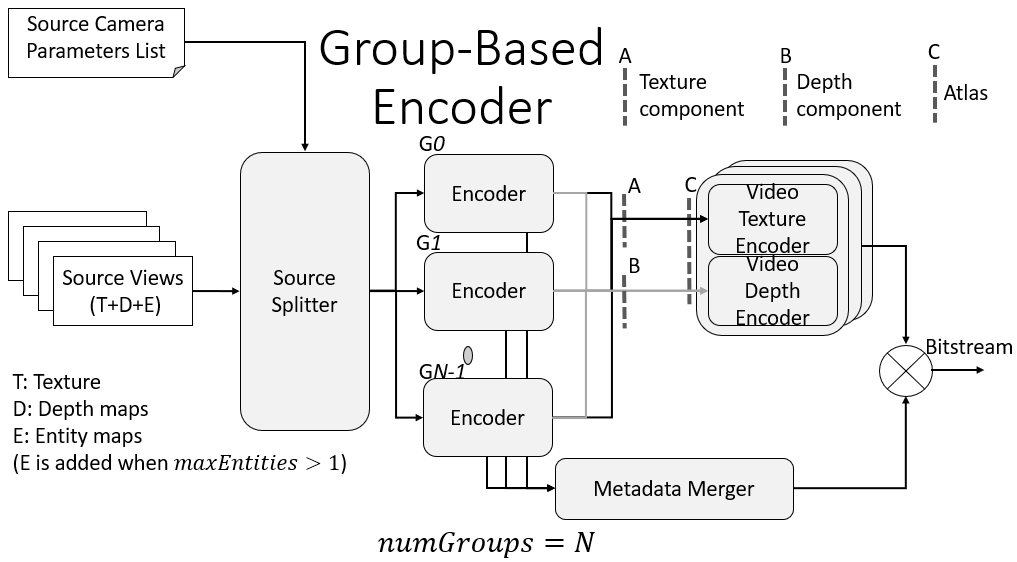


Figure : Processing flow for the group-based Encoder

The processing flow of the (per group) TMIV encoders (Figure 2) includes the view optimizer, the atlas constructor, and the depth occupancy coder. Most of the work happens in the atlas constructor which by itself has pixel pruning, mask aggregation, patch packing and atlas creation stages. When grouping is disabled, Figure 2 effectively becomes the top-level encoder stage diagram.

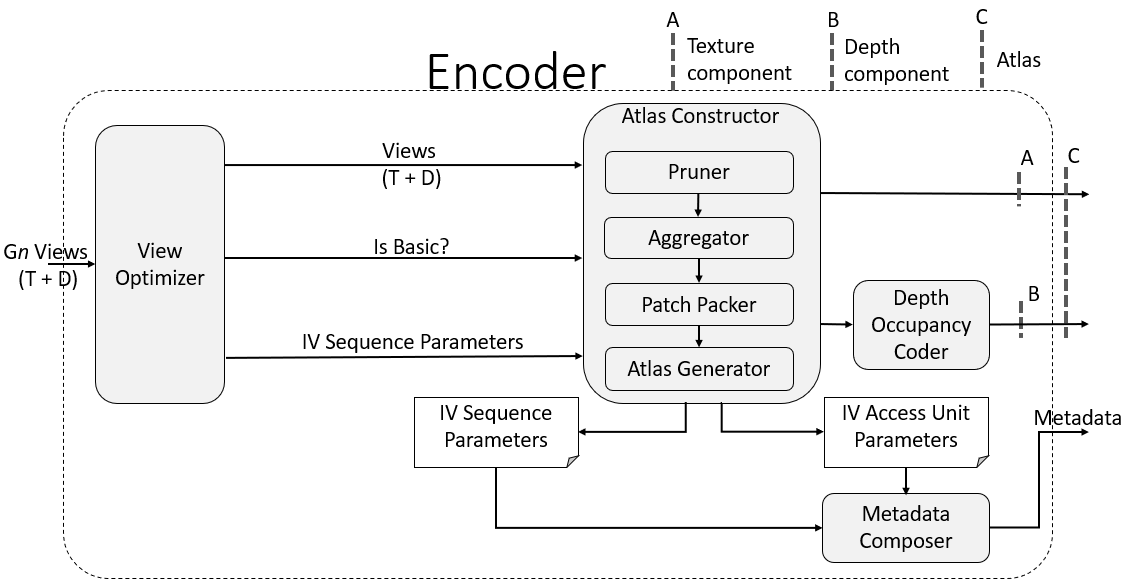


Figure : Processing flow of the (per-group) TMIV encoder

The TMIV decoder (Figure 3) consists of the video decoder & metadata parser, the atlas patch occupancy map generator, and the group-based renderer. There is also a variant of this renderer called the multi-pass renderer which is more suitable for atlases that contain only basic views.

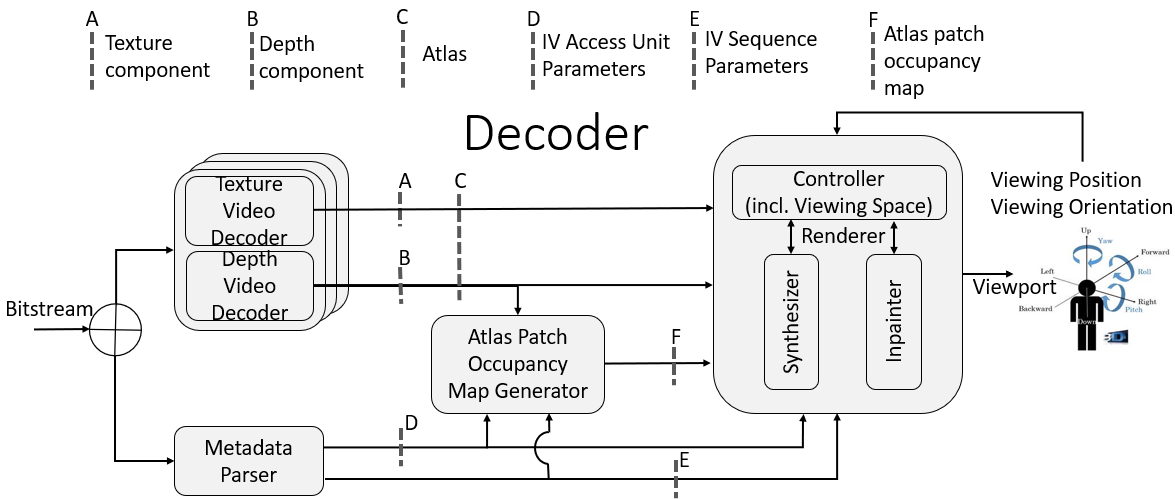


Figure : Process flow for the TMIV decoder

A future release of TMIV 3 is planned to include an entity-based encoder (Figure 4). This encoder adds an entity separator to create entity layers and an entity-based atlas constructor that independently prunes and aggregates each of the entity layers. It will be possible to combine entity-based and group-based encoding. The entity in the upcoming release of TMIV is based on objects as a show case.

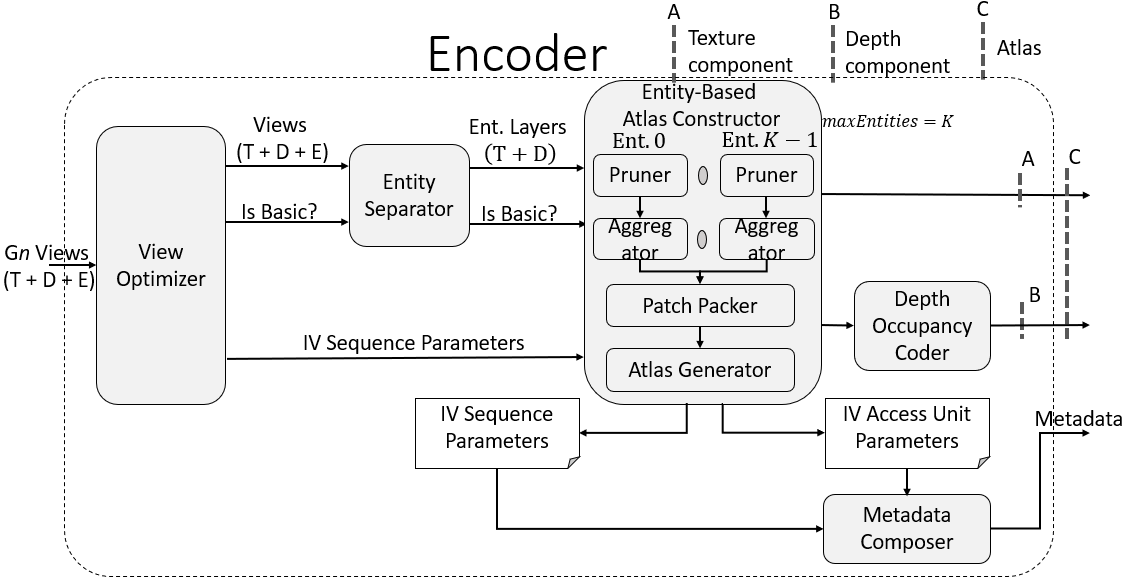


Figure : Process flow for the entity-based TMIV encoder

Figure 5 and Figure 6 further illustrate how data is processed across key components; the atlas constructor at the encoder side and the atlas patch occupancy map generator[[1]](#footnote-1) and the renderer at the decoder side. The figures demonstrate an example of how to map between 3 view representations and 2 atlases. More details are given in the related Sections 4 and 5.

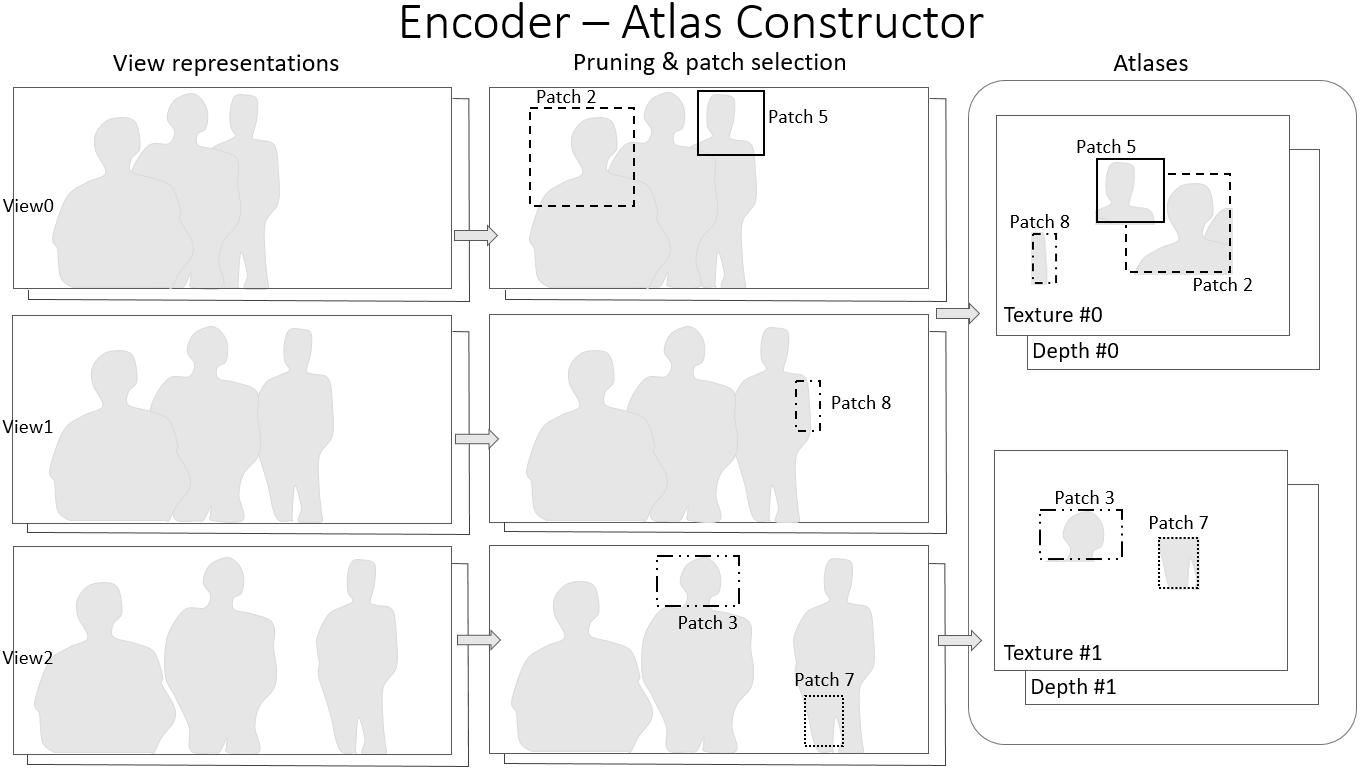


Figure : Data flow through key components within the TMIV encoder

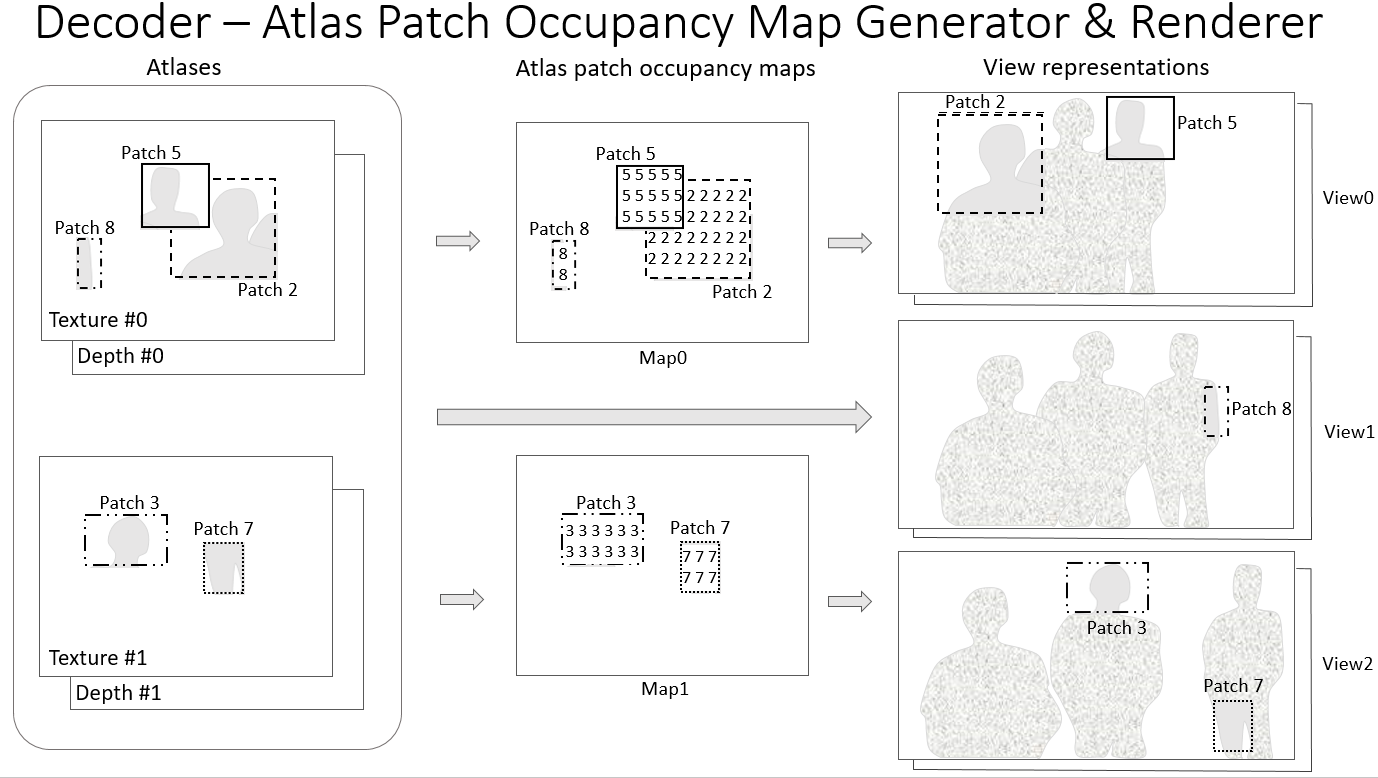


Figure : Data flow through key components within the TMIV decoder

## IO formats and execution

### Encoder inputs

Source views (texture & depth) representing projections of a 3D real or virtual scene are made available as inputs to the TMIV encoder. The source views can be in equirectangular projection (ERP), or perspective projection (PSP). They are provided in luma & chroma ~~4~~:2:0 format with 10 bit for texture and 8...16 bit for depth. For entity-based coding, an entity map at luma resolution is added per source view, indicating the entity that each pixel belongs to. An example of a set of input source views is illustrated in Figure 7. The source camera parameters list is provided in JSON[[2]](#footnote-2) format and includes the extrinsic parameters (x, y, z positions and yaw, pitch, roll orientations in the format defined by OMAF), the intrinsic parameters (focal lengths, principal points, and distortion coefficients), in addition to the projection type per source view.

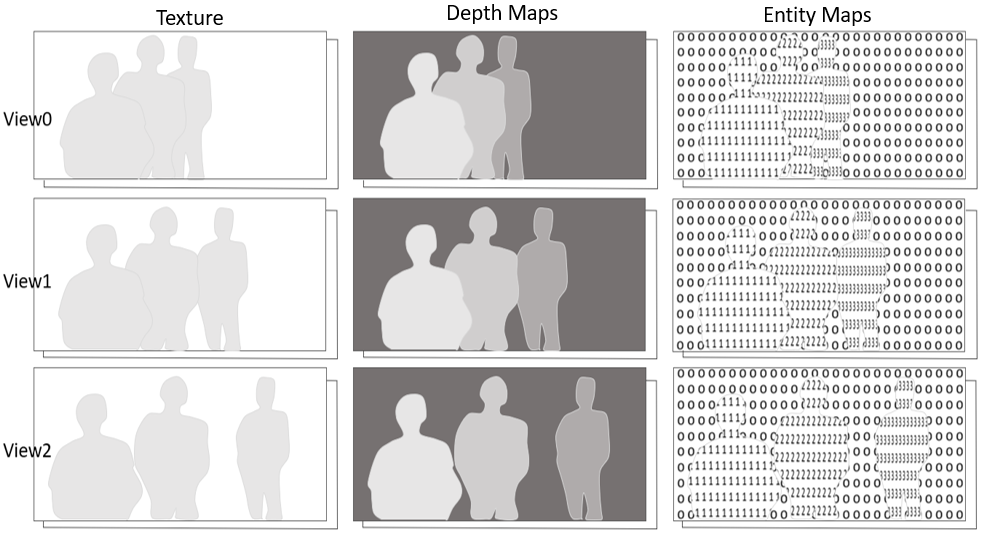


Figure : Input source views composed of texture views, depth maps, and entity maps

### Bitstream

The bitstream contains:

* An HEVC bitstream containing one or more layer pairs (as defined in [2]), each layer pair representing an atlas.
* Metadata composed of:
  + IV sequence parameters including a camera parameters list.
  + IV access unit parameters including an atlas parameters list.

### Decoder outputs

The output of the TMIV decoder is a perspective viewport or omnidirectional view according to a desired viewing pose, enabling motion parallax cues within a limited space. The rendered output is provided in luma & chroma ~~4~~:2:0 format with 10bit for texture and 16bit for depth. It can be displayed on either head mounted display (HMD) or on regular 2D monitor with tracking system feeding the updated viewer’s position and orientation back to the renderer for the next target view.

### Encoding atlases with only basic views

TMIV-SW includes a trivial view optimizer called the NoViewOptimizer that marks all source views as 'basic'. This effectively disallows the atlas constructor from pruning any of the pixels and thus the atlases will contain all of the basic views completely (typically one view per atlas). This mode helps evaluating the rendering in isolation of the other TMIV components.

### Configuration file generator

The TMIV-SW executable files take a configuration JSON file as input. There are no example configuration files, but TMIV-SW includes a script *make\_tmiv\_configs.py* that generates all CTC anchor configuration files. The script is designed to be easy to modify or import and create configuration files for new experiments (e.g. The *qp\_tuning\_configs.py* is an example of that). The output will be a directory structure with a configuration file for each encoding and decoding task including pose traces. There will also be HM configuration files with the correct atlas sizes and number of frames.

### Command-line options

To run the encoder or decoder, simply pass a configuration file:

*Encoder -c Configuration.json*

Top-level parameters such as startFrame or OutputDirectory or OutputCameraName maybe overriden using the -p command-line option:

*Decoder -c Configuration.json -p OutputCameraName v7*

It is possible to run the Decoder from within the Encoder (no HM) by specifying the reconstruct flag. This is useful for development and testing:

*Encoder -c Configuration.json -p reconstruct true*

## Coordinate systems, projections, and camera extrinsics

This section is derived from the RVS manual [4].

### OMAF coordinate system

The TMIV world coordinate system is that of MPEG-I OMAF[[3]](#footnote-3).

* points forward (the reference direction for a viewer),
* points left,
* points up,

Hereby is the notation for Cartesian unit vectors such that. For an untransformed camera the origin is the cardinal point.

The definition of image coordinates is:

* The top-left image corner is (0, 0),
* The top-left pixel center is at (½, ½),
* points right,
* points down.

Image positions are notated as .

### Perspective projection

Perspective projection requires an intrinsic matrix:

(1)

where all variables are in pixel units.

**Unprojection**

Taking into account the change of coordinate system, the projection equation is

(2)

where is the image position in pixel units.

**Projection**

The matching projection equation is

(3)

where is depth in meters and is the world position in meters. Please note that depth is typically stored as normalized disparities based on a configurable depth range, however in above equation is a length in meters.

### Equirectangular projection

For equirectangular projection the image is mapped on a horizontal angular range and vertical angular angle as specified in the JSON content metadata file.

**Unprojection**

For an image size, the spherical coordinates are:

(4)

(5)

The *ray direction* is:

(6)

and the world position is:

(7)

Whereby is the *ray length* which is the equivalent of depth for perspective projection. Please note that also ray length is stored as normalized disparities based on a configurable ray length range, however in the above equation is a real length.

**Projection**

The ray length and ray direction are trivially determined as

(8)

(9)

making use of the fact that valid ray lengths are

Finally, spherical angles are then estimated from:

(10)

(11)

Please note that the only difference between equirectangular projection and other omnidirectional projections is the mapping between spherical coordinates and image coordinates.

### Camera extrinsics

For MIV test materials the extrinsics are directly read from the JSON metadata files as a position and rotation vector. The Euler angles corresponding to yaw, pitch and roll respectively are internally converted to a rotation matrix according to:

Pose traces are comma-separated value files with the same six columns as the CTC tables and JSON metadata files: X, Y, Z, Yaw, Pitch, Roll.

The two rotations and two translations to transform from an input camera to a virtual camera are combined into a single affine transformation: x 🡪 Rx + t.

# Description of encoder-stage processing blocks

## Source splitter

Source views can be divided into multiple groups, so each group is processed separately. The grouping forces the atlas constructor to output local coherent projections of important regions (e.g. belong to foreground objects or occluded regions) in the atlases leading to significant improvements in the subjective and objective results especially for the natural content or at high bitrate levels. An automatic process is implemented to select views per group. It takes source camera parameters as input along with the number of groups *numGroups* as a preset from the config file and outputs views list to be included in each group. The source views are being distributed accordingly in multiple branches each has view optimizer and atlas constructor to process each group in parallel (i.e. independent of each other).

The soure splitter operates as follows; at the beginning, a camera pool including all available source views is formed and the number of cameras per group is set (by dividing the number of available source views to *numGroups* defined as a preset). Then camera parameters list is used to identify the range the cameras are spanning in X, Y, Z coordinates (as defined in OMAF coordinate system). The dominant range is selected as a basis to set key positions and distances of views available to these key positions are computed. Afterward based on the number of cameras for this group, the closest cameras to the first key position (located at the max camera position of the dominant axis across cameras in the camera pool) are selected and removed from the camera pool. Then a second key position is identified and the process is repeated covering the distribution of all source views across the chosen number of groups.

## Metadata merger

Each encoder (i.e. group’s encoding stage) produces metadata with its own indexed atlases or views and IV sequence parameters (i.e. group’s camera parameters list) in addition to the associated groupIds (set per atlas) within IV access unit parameters. In order to enable the renderer to interpret the IV access unit parameters and map the patches correctly across all views, the merger adjusts atlasId and viewId per patch and combines groupId parameters from all groups in the IV access unit parameters to account for different groups’ metadata. The merger also combines the groups’ IV sequence parameters so a complete list is formed following the correct viewId ordering.

## View optimizer

The view optimizer selects one or several views from the source views and labels them as basic views while the other non-selected source views are labeled as additional views (i.e. non-basic). It includes two steps:

* Determination of the number of basic views needed, considering direction deviation, field of view, and distance and overlap between views.
* Selection of the basic views, considering the distance to a central view position and some overlap.

The input of the view optimizer is the source views and the source camera parameters list (the position [x, y, z], the orientation (yaw, pitch, roll) [, , ], and the projection type). The output of the view optimizer is the view(s), their IV sequence parameters, and the Is Basic? vector flag indicating the basic views within the list.

### Determination of the number of *basic views*

First, the goal is to find a pair of views (view m, view n) that has the largest direction deviation according to the equation , where *i* and *j* are the indices the source views 0, with , and:

as shown in Figure 8.



Figure : Explanation of the directions deviation.

When two pairs provide the same maximum direction deviation, the view pair which has the largest sum of field of views (FOVs) is selected:

When two pairs provide same maximum sum of FOVs, the view pair which has the largest distance between each other is selected:

Second, the overlap between the two views is computed, as illustrated in Figure 9.


Figure : Illustration of the overlap and its calculation.

Each pixel position (i, j) of the view m is projected on the view n in position (i', j'). The weighted sum of overlapped pixels whose new position (i', j') is in the FOV of view n is computed as in the equation below, with meaning that (i, j) is visible by both view m and view n:

where is the spherical weight of each pixel position (i, j). FOV is in Steradian unit.

Finally, the number of basic views is determined:

* If , only one basic view is selected.
* If overlap , multiple basic views including view m and view n are selected.

Please note that if number of groups *numGroups* set in the config file is bigger than 1, then at least one basic view is included per group.

### Selection of the basic views

#### When only one basic view is needed, according to step 1, the following applies:

The source view that has the largest FOV is selected as a basic view . If several views have the same largest FOV then the following applies:

* Calculate the central camera position of the source capturing system given the source camera parameters list.
* Select as a basic view the source view which camera position is the closest to the central camera position:

#### When several basic views are needed, according to step 1, the following applies:

The views m and n found in step1 are selected as basic views. The view k which has the largest direction deviation with view m and view n is determined:

If view k has less than 50% FOV overlap with the already selected basic views m and n, then view k is selected as a basic view, and the same process is repeated to find the next basic view. Otherwise the process stops.

All other non-selected source views are labeled as additional views and passed along with the basic views to the output of the view optimizer.

## Atlas constructor

The atlas constructor takes as input the views, the Is Basic? vector flag along with the associated IV sequence parameters and output atlases, with the IV sequence parameters and with the IV access unit parameters, as shown in Figure 2. Each basic view is carried in the atlas as a single, fully occupied patch (assuming the atlas size is equal or larger than the basic view size otherwise the basic view may be split into multiple atlases). The additional views are pruned into multiple patches which may be carried along with a basic view’s patch in the same atlas if the atlas is of larger size or in separate atlas(es).

The atlas constructor is composed of four configurable parts: the pruner, the aggregator, the patch packer and the atlas generator. The pruner and the input of the aggregator operate at the frame level. The output part of the aggregator and the patch packer operate at intra-period level.

Note that the Pruner, the Aggregator and the Patch packer blocks only process the depth component, as illustrated in the detailed block diagram drawing of the Atlas Constructor.

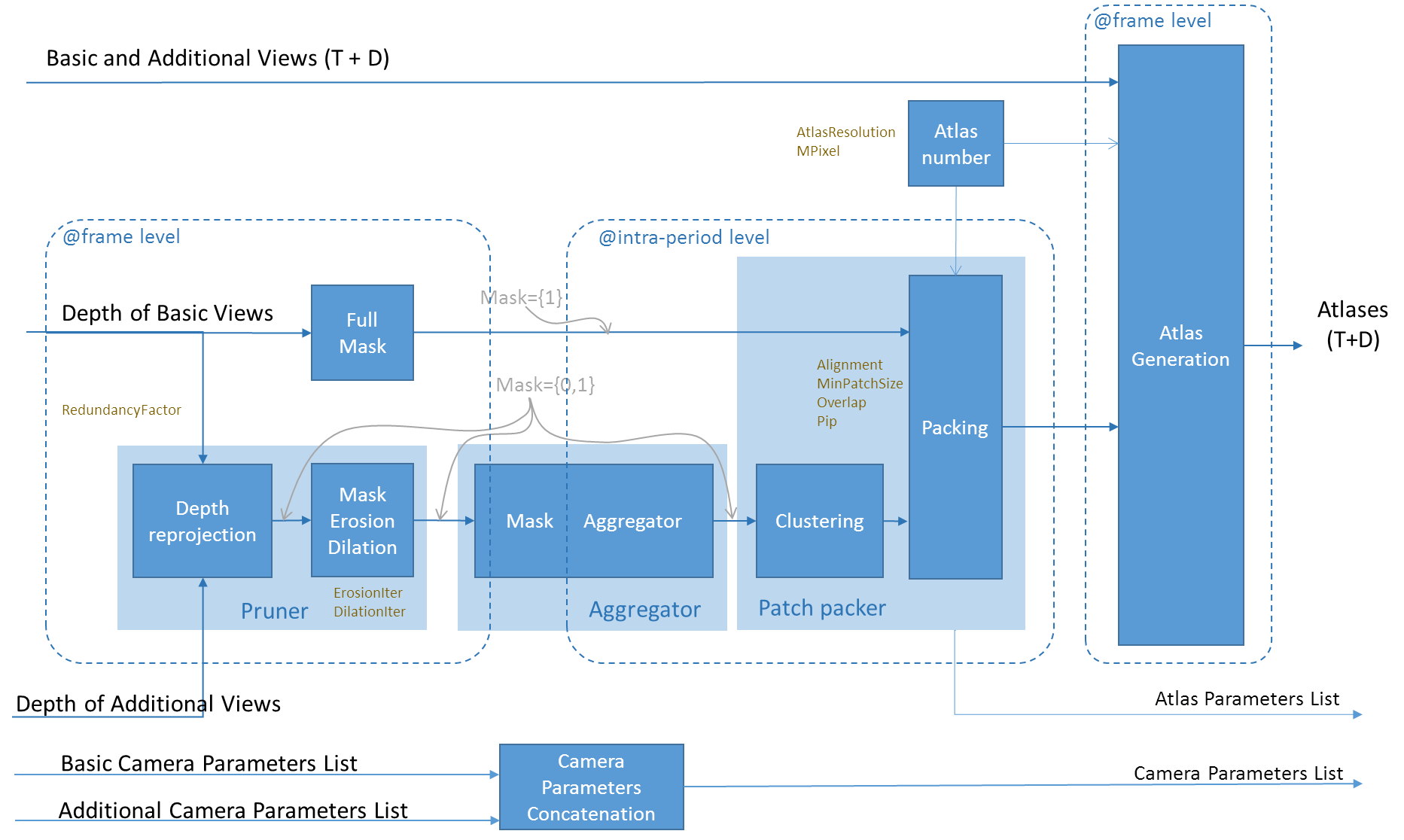


Figure : Detailed block diagram of the Atlas Constructor

## Hierarchical pruner

A multiview representation of a scene inherently has interview redundancy. The purpose of the Hierarchical Pruner is to indicate which areas of the transport views may be safely pruned. The pruner operates on a per-frame basis, receiving IV sequence parameters (IVSP) and a multiview plus depth (MVD) frame, and outputting a multiview (MV) mask of the same size.

For additional views, mask values are either 'pruned' (0) or 'preserved' (255). For basic views all pixels are 'preserved'.

The method has been devised with the following goals in mind:

* Remove redundancy between all pairs of views,
* Prefer fewer larger patches,
* Temporal consistency,
* Realistic complexity.

To achieve these goals, the pruner creates a hierarchy of views in a greedy fashion. The hierarchy (Figure 11) serves the first two goals. Temporal consistency is maintained by establishing the hierarchy only when the view parameter list changes. (For the CTC this is only at the first frame.) The greedy nature of the algorithm keeps the complexity at bay.



Figure : Pruning hierarchy for one basic and three additional views. The order of the partial views is established at frames for which the view parameter list changes. Each arrow is a trial synthesis operation using only the preserved pixels.

### Incremental synthesizer

The Synthesizer that is part of the Renderer is repurposed within the Hierarchical Pruner but used in a more interactive way. Instead of directly synthesizing a full MVD frame, the Incremental Synthesizer renders only one view at a time and is able to provide intermediate synthesis results (without inpainting). This allows the pruner to accurately determine the interview redundancy.

To speed up synthesis the incremental synthesizer constructs a textured mesh for preserved pixels only. The mesh has one vertex per pixel for basic views and much less vertices for additional views.

### Establishing the pruning hierarchy

On the first frame the following algorithm is used with above mentioned inputs and outputs:

Initialization:

1. For each additional view, an incremental synthesizer is initialized.
2. Each basic view is synthesized to each additional view.

While there are synthesizer instantiations:

1. For each instantiation, create the pruning mask.
2. Select the instantiation with the maximum number of preserved pixels.
3. Keep the associated pruning mask for output.
4. Synthesize the preserved regions to the other instantiations.
5. Remove the selected instantiation.

Note that the preserved fraction is typically quite low for additional views and this provides a significant reduction in computational complexity. A typical case would be 100% (basic), 20% (1st additional), 10%, 5%, etc. These statistics are output for analysis purposes.

### Applying the pruning hierarchy

For subsequent frames the following algorithm is used with similar complexity.

Initialization:

1. For each additional view, an incremental synthesizer is initialized.
2. Each basic view is synthesized to each additional view.

For each instantiation in predetermined order:

1. Update the pruning mask.
2. Keep the pruning mask for output.
3. Synthesize the preserved regions to the other instantiations.
4. Remove the instantiation.

### Pruning mask creation

The pruner uses two criteria to determine if a pixel may be pruned:

* The pixel should be synthesized by the views higher up in the hierarchy,
* The ratio between synthesized and source depth should be less than a threshold.

A mask typically has holes and irregularities which are cleaned up by a classical iterative erosion and dilation method on a 3x3 structuring element:

* For the erosion, a pixel that has at least one empty neighbor is discarded (pixel = 0).
* For the dilation, a pixel that has at least one non empty neighbor is filled (pixel =1).

## Aggregator

The mask is reset at the beginning of each intra period. Then, an accumulation is done for each mask’s pixel i with the 1 value across the different frames of the intra period by implementing the logical operation OR as follows:

aggregatedMask[i]@current\_frame =

max(Mask[i]@current\_frame, aggregatedMask[i]@previous\_frame)

The process is completed at the end of the intra period by outputting the last accumulation result. Figure 12 illustrates for a pruned view at frame i, the accumulation of non-null samples (drawn in white) between the frame i and frame i+k within an intra period; it can be seen that contours are getting thicker on the changing part of the depth map accounting for the motion within the scene.

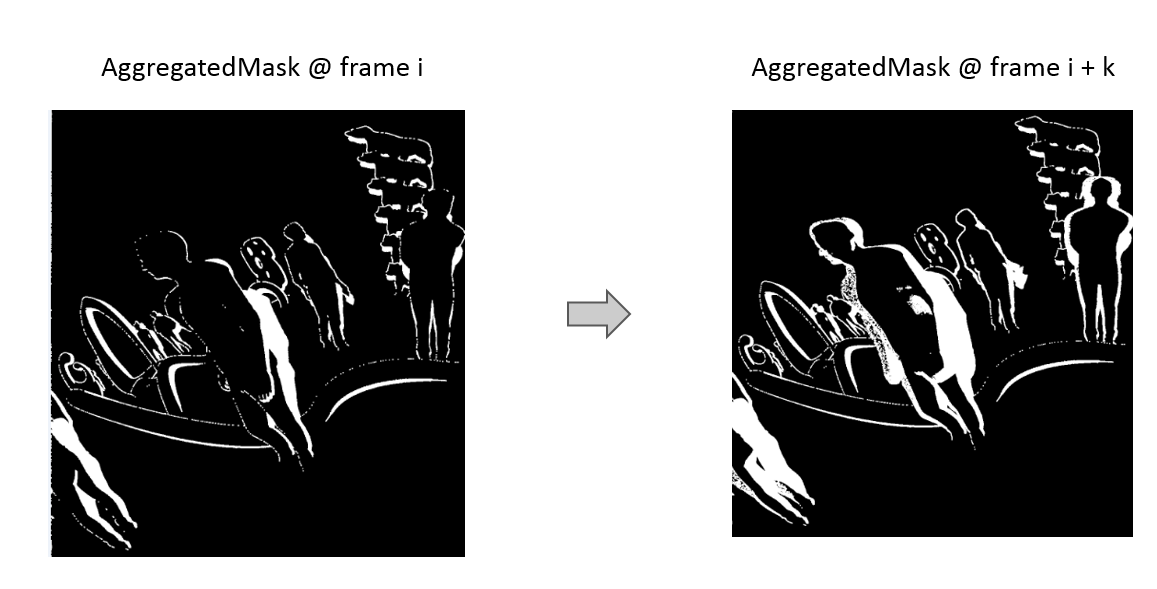


Figure . AggregatedMask evolution within an intra period.

## Patch packer

The packing process consists of three steps that are described in subsequent sections:

1. Clustering (Section 4.7.1)
2. Atlas allocation (Section 4.7.2)
3. Packing (Section 4.7.3)

### Clustering

This block is in charge of identifying what is called “clusters”. A cluster is a rectangle, containing a set of connected mask pixels of 1s value obtained by a region growing process. The connection criteria of one pixel is the presence of at least one other pixel among the 8 neighbors.

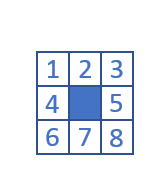


Figure : 8-pixel neighborhood for defining the connectivity criteria for region growing.

An example of the clustering is illustrated in Figure 14 where each cluster of an already pruned view is represented by a specific false color. The parameters associated to each cluster are:

* x and y positions of the top left rectangle corner.
* Width and height of the rectangle.
* The cluster are then sorted by a decreasing size order.

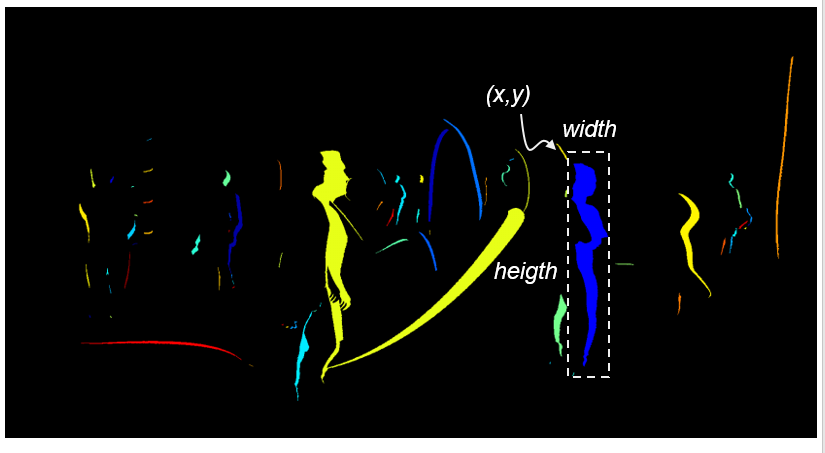


Figure : Clusters represented in false color on a pruned view

### Atlas allocation

This function defines the number of atlases per group by 2 input parameters passed in the configuration file:

* "*AtlasResolution*": [4096, 2048] # [Atlas width, Atlas Height]
* "*MaxLumaSamplesPerFrame*": 33554432 # Maximum size of all atlases combined expressed in luma samples per frame (for texture plus depth)

The number of atlases is given by:

### Packing

The packing process tries to sequentially pack each cluster into the atlases. The input parameters are the following:

* *“Alignment”* is defined as a number of pixels. Patch size and patch position are multiple of alignment. Default value is 8.
* *“MinPatchSize”* is the number of pixels of the smallest edge of the patch, below which the patch is discarded. Default value is 8.
* *“Overlap”* is the number of pixels which will be added to a frontier of a newly split patches; it prevents seam artefacts. Default value is 1
* *“PiP”* is a flag enabling the Patch-in-Patch feature when equal to 1. Default value is 1.

The packing process is based on a version of MaxRect algorithm [6]. It makes use of the existing “Used Space” first, by examining the space which is efficiently occupied (“Filled space”). It is made of intricated loops which are described by the following pseudo-code:

|  |
| --- |
| *For each cluster*  *For each atlas*  *Try to push the cluster in Used Space*  *Try with 0° rotation first*  *Else with 90° rotation*  *Else*  *Try to push the cluster into free space*  *Try with 0° rotation first*  *Else with 90° rotation*  *Else*  *Split the cluster into 2 parts by its largest border*  *For each resulting 2 parts*  *If smaller than MinPatchSize*  *Discard*  *Else*  *Put the part in the cluster priority list* |

Currently (as described in the pseudo-code), only a possibility of 90o rotation (with no flipping) is implemented in TMIV2.0 encoder although the metadata allows more variations and TMIV2.0 decoder can interpret all possibilities.

The output of the block is the patch list for each atlas with all necessary information to recover at the decoder side:

* The location in the atlas patch\_pos\_in\_atlas\_x and patch\_pos\_in\_atlas\_y along with the AtlasId.
* The location in the original view representation patch\_pos\_in\_view\_x and patch\_pos\_in\_view\_y, and its dimensions patch\_width\_in\_view and patch\_height\_in\_view.
* The related ViewId, which itself refers to the de-projection parameters for that view in the decoder
* The entityId in case *maxEntities* > 1 (see section 4.11 for details)
* A possible rotation by i\*90° where i = 0, 1, 2, 3
* A possible vertical flip

The packing operation from view representation to Atlas is done with rotation (first) then vertical flipping (second). The eight achievable states for the patch packing are covered by the “rotation/vertical flip” configurations.

Note that at the encoding side, the rotation of 90° is here meant to be from view representation to Atlas and is counter-clockwise, i.e. rotates the Y-Axis on the X axis, as illustrated in the following figure (Figure 15).

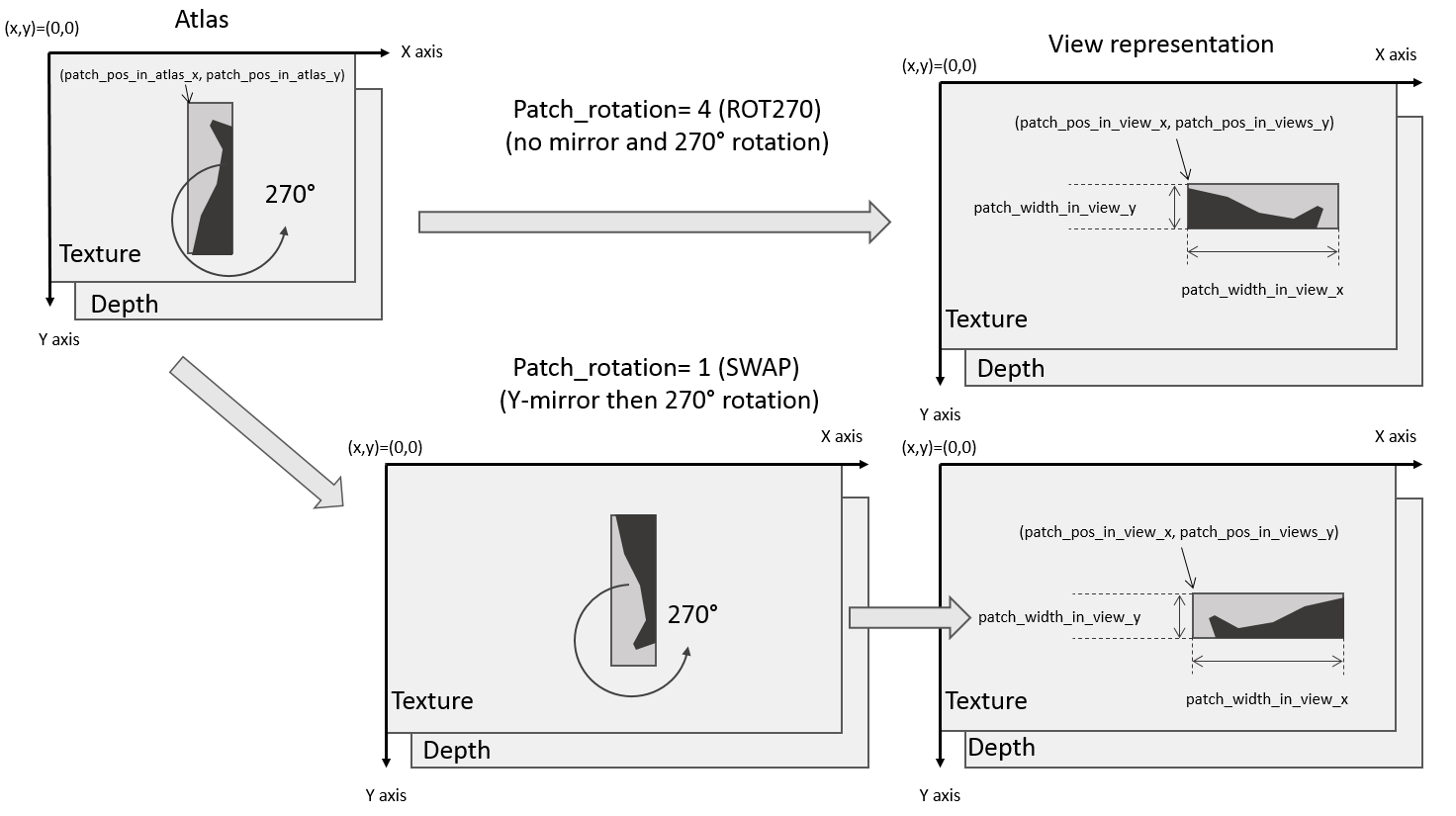


Figure : Meaning of patch related parameters.

Although the working draft [2] allows rectangular patches to have unoccupied pixels, in TMIV3.0 the entire rectangular patch is currently set to be occupied. Unoccupied pixels are indicated by setting the corresponding depth pixel luma values to 0.

## Atlas generator

The final operation within the atlas constructor is writing the patches in the buffer allocated to the atlas (both the depth and the texture components). Figure 16 illustrates the generation of an atlas, with the successive write of patch 2, 5 and 8. While the packing algorithm is using the information of samples that are mandatory and are non-pruned (represented by area inside the perimeters in dash), the copy of the patch is rectangular, resulting in a heap of possibly overlapping rectangles.

These rectangles are fully occupied in general. They may be partially empty when basic or additional views contains invalid pixels (it may be the case when these latter views are not source views for instance). In that case, the null value in the depth expresses the invalidity of a sample.

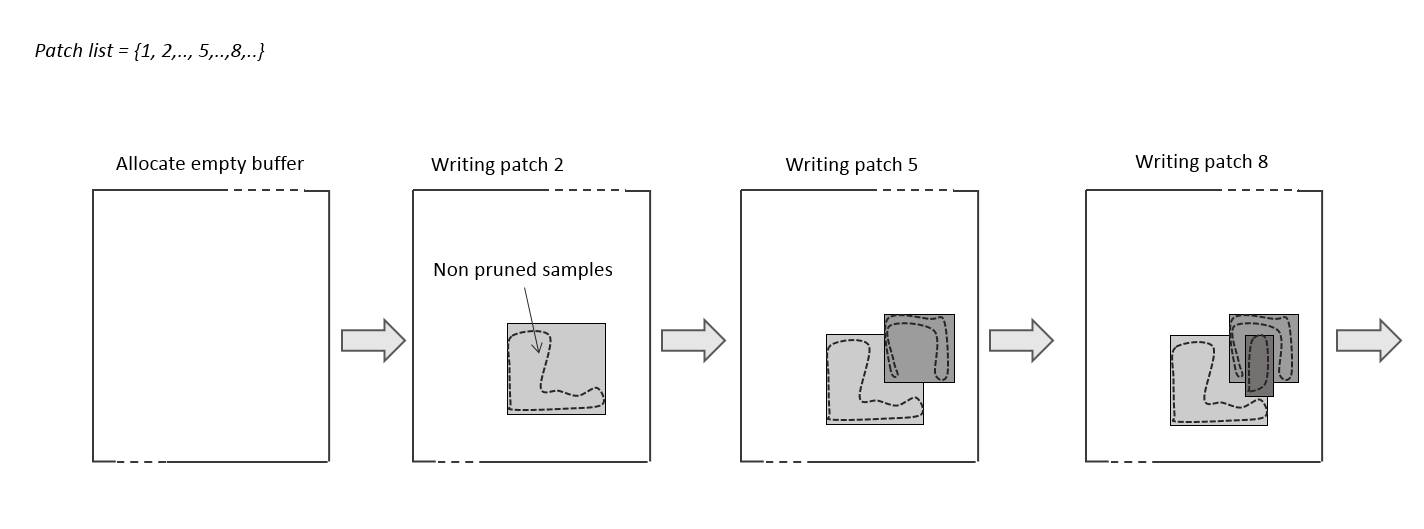


Figure : Successive writing of patches into an atlas.

## Depth occupancy coder

### Introduction

WD3 [2] specifies how to encode occupancy information within depth atlases. Without repeating too much of the WD3 depth decoding process, the decoding is based on a normalized disparity range, a depth-occupancy map threshold an optional clamping start value. These values are signalled per view or even per patch.

The TMIV decoder supports all cases. Assuming 10-bit full range depth atlases, the transformation may be described in pseudo-code as:

1: valid := x ≥ depthOccMapThreshold

2: if (valid) {

3: normDisp := max(kilometer-1,

4: normDisp0 + (normDisp1023 - normDisp0) \* (max(depthStart, x) 1023))

5: depth := 1 / normDisp

6: }

Line 1 is part of the atlas patch occupancy part generator (Section 5.2), lines 3...5 are effectively part of the Synthesizer (Section 5.7) and lines 2 and 6 are implicit in the TMIV decoder.

To summarize:

* Either an atlas value is "invalid/non-occupied",
* or it is a depth value in meters,
* There is an implementation-defined maximum depth value (1 km).

### TMIV encoder

The atlas constructor outputs rectangular patches with full occupancy so the occupancy coding capability of WD3 is not fully utilized by TMIV encoder. Because of this, the depth occupancy coder implements a simple method that recognizes two situations as depicted in Figure 17 and Figure 18:

1. When a source view has only valid depth values, depthOccMapThreshold is set to zero. This effectively encodes full occupancy (Figure 17).
2. When a source view has invalid depth values, depthOccMapThreshold is set to a configured value (T) and the normalized disparity range is adjusted such that the value 2T corresponds to the far depth (Figure 18).

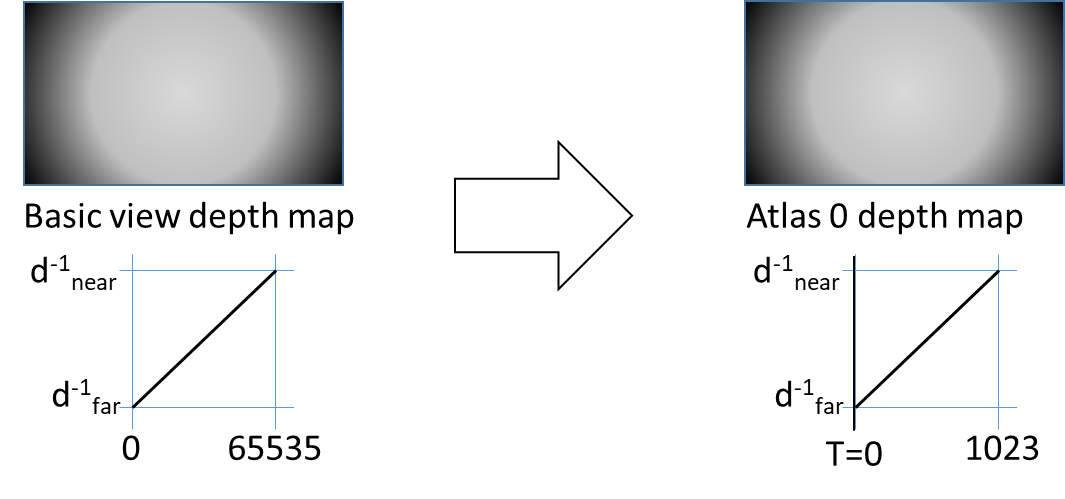


Figure : When the source material has only valid depth values, the depth occupancy coder only performs u(16) to u(10) scaling and the depth-occupancy map threshold is set to zero to signal full occupancy

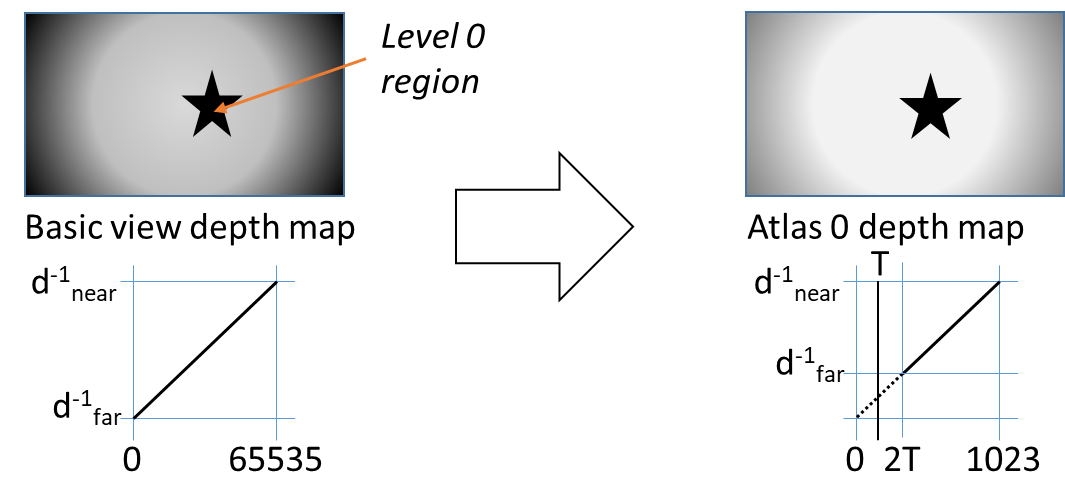


Figure : When the source material has invalid depth values, the depth occupancy coder not only performs u(16) to u(10) scaling, but it also sets the depth-occupancy map threshold to a configurated value (T) and the normalized disparity range is modified such that value 2T corresponds to the far depth

## Entity separator

A future version of TMIV 3 will support entity-based coding.

The entity separator, shown in Figure 5, is invoked when *maxEntities* > 1. It splits the (basics and additional) views (texture and depth maps) based on the associated entity maps into multiple layers where each layer includes pixels belong to a single entity. The “Is Basic ?” flag vector is carried to the entity layers to inform the pruner (within the entity-based atlas constructor) to perform the proper pruning operations.

## Entity-based atlas constructor

A future version of TMIV 3 will support entity-based coding.

The entity-based atlas constructor, shown in Figure 5, is similar to the regular atlas constructor. It runs the pruner, the aggregator, and the clustering on the entity layers belonging to the same entity while looping over all available entities. This results in patches that only include regions belong to a single entity, hence, the associated entityId per patch is updated to indicate what entity the patch was originated from. The patch packer then places all patches within the generated atlases.

## Video encoder

The HEVC encoder of profile Main10 in the Random Access configuration is used to encode the texture and depth of the atlas(es) video (in separate layers) provided in 4:2:0 10 bit format.

# Description of decoder-stage processing blocks

## Video decoder and metadata parser

The video decoder receives HEVC encoded atlases. The texture component and the depth component may be decoded independently by HEVC decoders.

The metadata parser splits the received metadata into IV sequence parameters and IV access unit parameters. Both are used by the Atlas Patch Occupancy Map Generator and the Renderer to produce the viewport requested (Figure 3).

## Atlas patch occupancy map generator

An occupancy map is generated for each atlas. It has the same size as the atlas. In the TMIV-SW the atlas patch occupancy map generator is only updated at each new intra period, while the signalization of sample invalidity is only used within the renderer.

This map gives for each sample the number of the patch that belongs to, as illustrated in the figure below. The map is simply created by browsing the atlas parameters list within IV access unit parameters from the parsed metadata exactly in the same order as during its creation, to resolve any overlapping. Figure 19 illustrates how three overlapping patches can be resolved by following the right order.

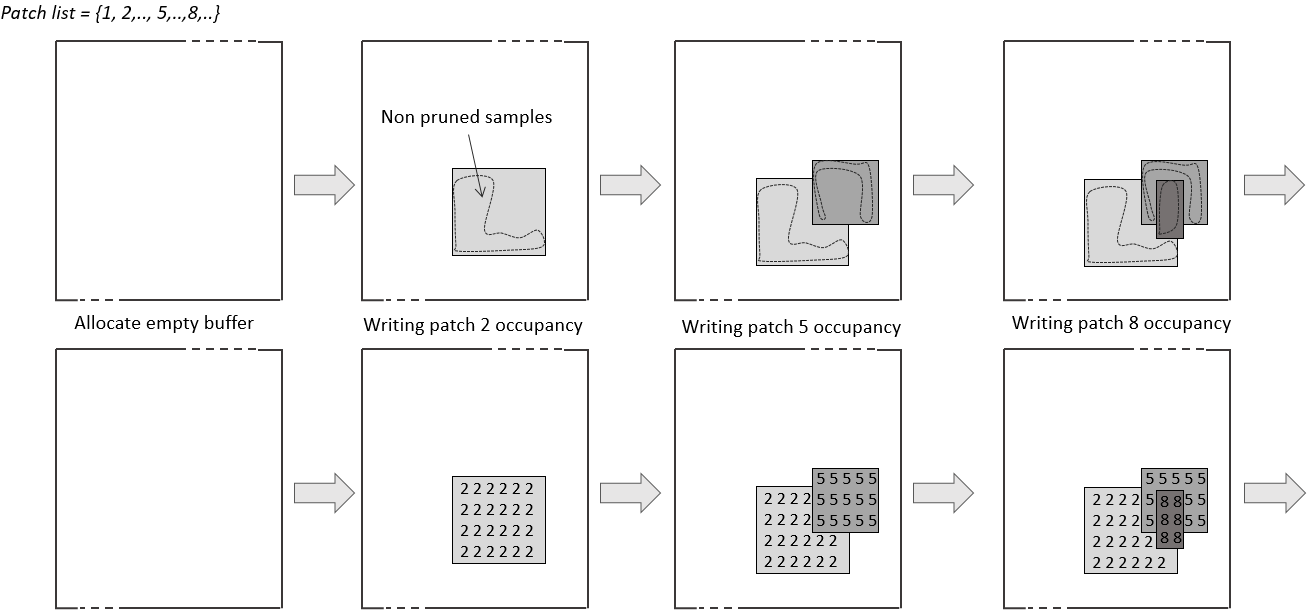


Figure : Atlas Occupancy Map Generation in an ordered manner to resolve the overlapping patches.

This map is then used in a loop on the atlas’s samples to get their respective PatchId, which itself enables getting the respective ViewId, hence enabling the de-projection and re-projection on the viewport.

To keep tracking the valid-pixels per patch while accounting for compression artifacts and noise during the streaming, we set to zero all depth values (in the depth component of the atlases) that are below the depth-occupancy map threshold as explained in Section 4.9.1. This results in identifying and excluding the non-valid pixels within patches during the rendering process which helps in suppressing their artifacts.

## Renderers

The TMIV-SW has a flexible rendering engine that is able to render directly from atlases using pipelined reprojection and parallel rasterization of triangles to reduce the wall time for generating viewports. Rendering is based on RVS with improvements from CfP responses and newer proposals. The output is a view with texture and depth of the same bit depth.

There are three renderers in TMIV: the group-based renderer (5.6), the multi-pass renderer (5.5) and the single-pass renderer (5.4). As depicted in Figure 3 these renderers have three parts:

* The renderers themselves are Controllers that accept input data, invoke the Synthesizer (possibly multiple times) and the Inpainter and forwards the output.
* The Synthesizer (§5.7) reprojects, rasterizes and blends the input data.
* The Inpainter (§5.8) replaces any missing pixels (indicated by level 0 in the depth map) with interpolated texture and depth data.

## Single-pass renderer

The single-pass renderer has a minimal controller that performs synthesis and inpainting. It serves as an example.

## Multi-pass renderer

The multi-pass renderer is mainly used for atlases that contain only basic views. The controller invokes the Synthesizer in multiple passes where *“NumberOfPasses”* and *“NumberOfViewsPerPass”* can be tuned as part of the configuration parameters.

At first only nearby views (or patches belonging to nearby views) are used for the synthesis to output coherent synthesis results. Then, the view selection is extended to include views further away (or patches belonging to views further away) from the target view to output more complete synthesis results. The process is repeated over the chosen number of passes. When operating on atlases, local occupancy maps are created per pass such that they include only the patchId of patches from the selected views per pass. Then they are passed to the synthesizer to render only these selected patches. Afterward, the synthesis results of individual passes are merged together in a successive manner to output a coherent and complete synthesis results. Finally, the Inpainter is engaged to fill the missing regions prior to outputting the requested target view. A block diagram of the multi-pass operation invoked by the Controller is shown in Figure 20.

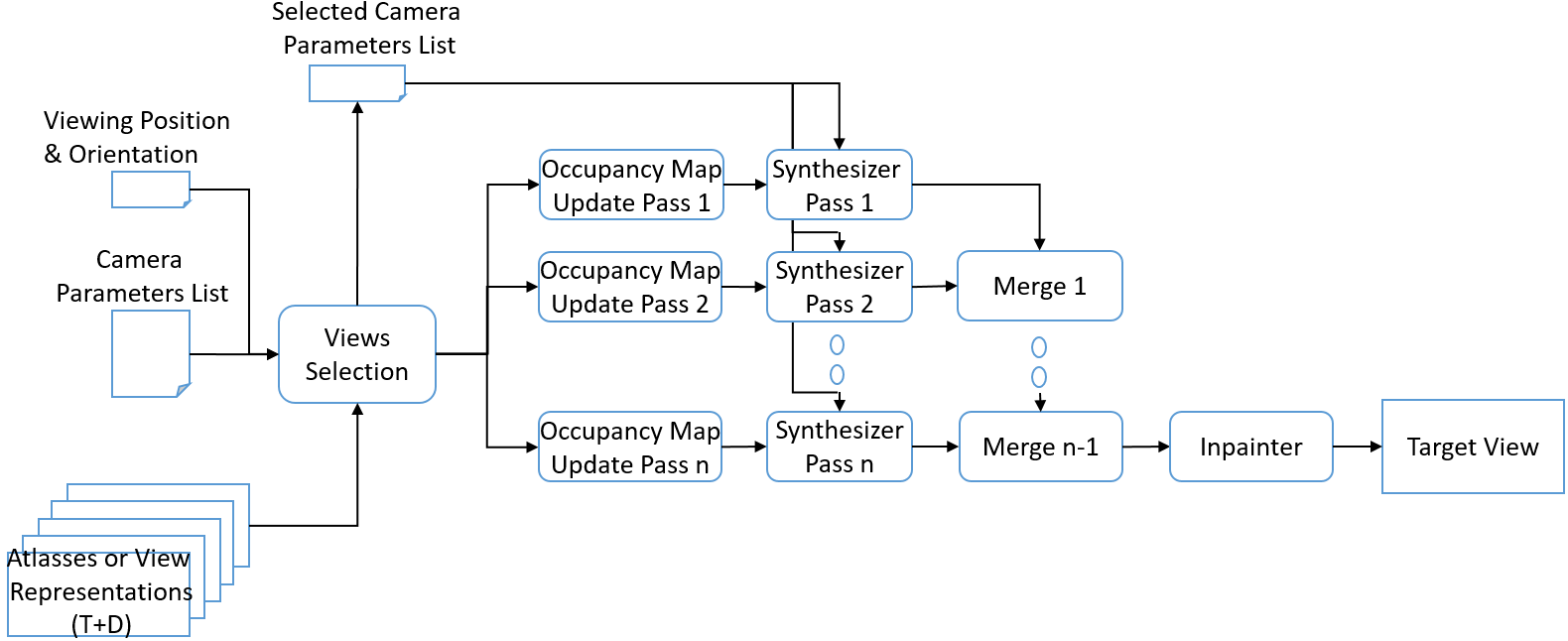


Figure : Process flow for the multi-pass renderer.

## Group-based renderer

The group-based renderer is capable of rendering from local patches within each group separately to produce significant quality improvement. The renderer’s process flow, illustrated in Figure 21, is composed of group selection stage, multiple passes each running the synthesizer with different set of atlases and output an intermediate view synthesis, and the merging stage to combine all intermediate synthesized views into a final desired viewport. The number of groups *numGroups* included in the metadata is used to set the number of passes required. Also, the selection of which atlases to be used for synthesizing intermediate view in a particular pass depends on their groupId parameters included in the metadata, the group distance to the desired pose being synthesizing for (e.g. that may be requested by a head-mounted display), and the pass index. In case *numGroups* = 1, then the group-based renderer converges to the simple renderer.

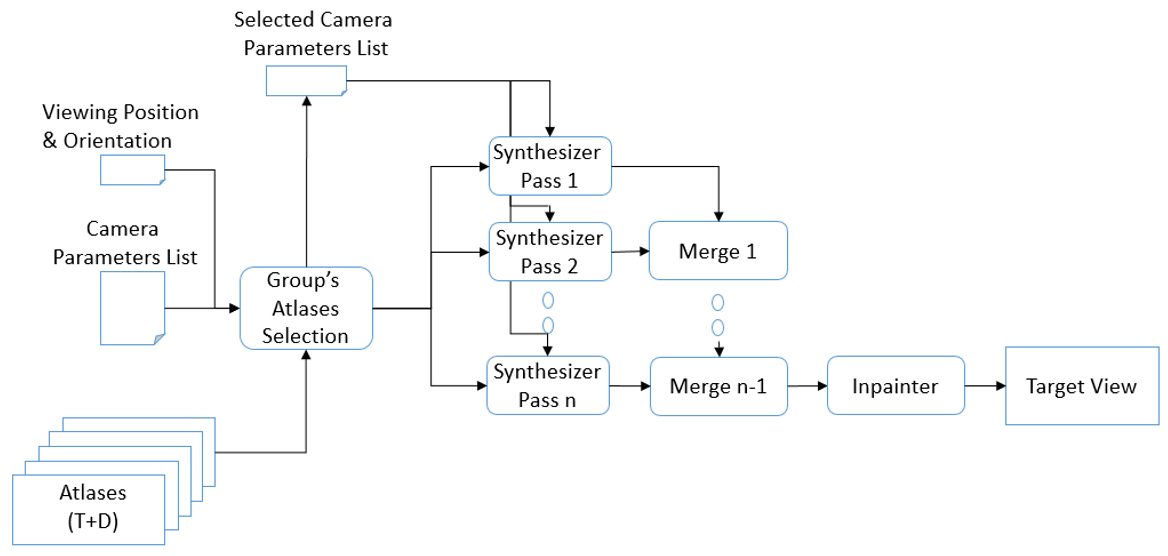


Figure : Process flow for the group-based renderer.

To avoid having large impact on the pixel rate when *numGroups* > 1, a limited number of atlases per group (at least one has to be a basic view) is desired. Depending on the content motion/occlusions, there may not be enough room to carry all necessary patches within the atlases. The goal is to let the group-based renderer try to pick up these missing patches from other groups during the merging stage while maintaining the coherent patches generated from the lower pass synthesis (done within the same group). During the merging step, a depth check is introduced and the merging is done as follows:

* + If the pixel *i* from the lower pass synthesis has normalized disparity value *id* that is larger (or equal) than the normalized disparity value *jd* of the exact pixel *j* in the higher pass, then carry the pixel *i* from the lower pass into the merged view.

*if (id >= jd ), return i.*

* + If the pixel *i* from the lower pass synthesis has normalized disparity value *id* that is smaller than the disparity value *jd* of the exact pixel *j* in the higher pass, then there is a conflict! To resolve, the value is carried from front objects in case the *depthLowQualityFlag* is false. Otherwise, the value is carried from the lower pass synthesis if existed (similar to what is done in the multi-pass renderer). The *depthLowQualityFlag* is read directly from a json config file and is computed as illustrated in Annex 3.

*if(id < jd){  
 if(depthLowQualityFlag* >0*)  
 return i; // Always copy from lower pass (traditional multi-pass approach)   
 else  
 return j; // Always copy from the front objects  
 }*

## Synthesizer

### Introduction

Like RVS [4], the synthesis is based on:

1. Generic reprojection of image points,
   1. Unprojection image to scene coordinates (using intrinsics source camera parameters),
   2. Changing the frame of reference from the source to the target camera by a combined rotation and translation (using extrinsics camera parameters),
   3. Projecting the scene coordinates to image coordinates (using target camera intrinsics).
2. Rasterizing triangles,
   1. Discarding inverted triangles,
   2. Creating a clipped bounding box,
   3. Barycentric interpolation of color and depth values,
3. Blending views/pixels.

While RVS was designed to render full views, the Synthesizer works with arbitrary vertex descriptor lists, vertex attribute lists, and triangle descriptor lists (which is very much like OpenGL). The view blending is per pixel and independent of the rendering order. It is thus possible to render any triangle from any patch in any order.

### Rendering from atlases

As part of the decoder (primary purpose) the renderer takes as input:

* Multiple atlases with 10-bit texture and 10-bit depth (normalized disparities),
* An atlas patch occupancy map per atlas with 16-bit values (Figure 19),
* IV access unit parameters including an atlas parameters list,
* IV sequence parameters including a camera parameters list,
* Target camera parameters for a perspective viewport or an omnidirectional view.

The output of the renderer is a single view (viewport or omnidirectional) with 10-bit texture and 10-bit depth components.

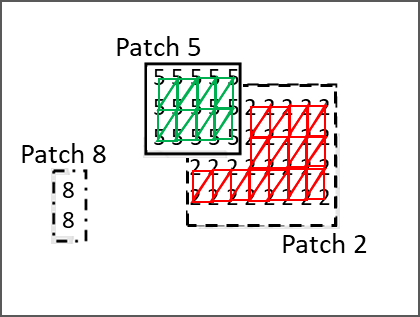


Figure : Creating a mesh from an atlas. Triangles between pixels from Patch 5 and 2 are omitted. Note that Patch 8 is not drawn because no triangle can be formed. Unused pixels are skipped too.

The process is to build a mesh (Figure 22) from each of the atlases:

* The **vertex descriptor list** is formed pixel-by-pixel:
  + Skip or write dummy values for unoccupied pixels (occupancy value 0xFFFF),
  + Looking up the atlas parameters list using the PatchId in the occupancy map,
  + Looking up the camera parameters list using the ViewId in atlas parameters list,
  + Calculating the position of the vertex in the view.
  + Reprojecting from the source view to the target view.
* The **vertex attribute list** is simply the texture values converted to YUV 4:4:4.
* The **triangle descriptor list** is formed by:
  + For each pixel consider two triangles [ / ]
  + Add the triangle when all vertices have the same PatchId.

This mesh is then rasterized using barycentric interpolation of texture and depth. Multiple atlases will be utilized to render from directly in order to have an efficient pipeline for mesh generation and rasterization operations.

### Pixel blending

The blended value of a pixel component is the weighted sum over all pixel contributions. This choice enables pixel blending in arbitrary order. The weight of a contributing pixel is determined by multiplying three exponential functions with configurable parameters (Table 2).

The weighted sums are normalized by the depth weight to reduce the required internal precision. All three inputs (ray angle, depth and stretching) are computed in the reprojection process.

Table : Description of the blending process.

|  |  |  |
| --- | --- | --- |
| **Input** | **Description** | **Purpose** |
| RayAngle | The angle [rad] between the ray from the input camera and the ray from the target camera. | Prefer nearby views over views further away (soft view selection). |
| Reciprocal depth | The reciprocal of the depth value in the target view [diopter]. | Prefer foreground over background (depth ordering). |
| Stretching | The unclipped area of the triangle in the target view relative to the source view. | Penalize triangles that stretch between foreground and background objects. |

## Inpainter

In order to fill holes in the virtual view, a 2-ways inpainter is used. For each empty pixel with no information, two neighbors are being searched: the nearest non-empty pixel at the left and at the right. The color of the inpainted pixel is a weighted average of colors of the left and the right neighbor, weighted by the distances to these pixels. In the case of significant difference between depth of both neighbors, the color of the neighbor with further depth is copied instead of using a weighted average.

However, horizontal inpainting of the virtual view would cause appearance of unnaturally-oriented lines in the case of projecting ERP images to perspective views. Therefore, for ERP images the additional step of changing projection type is performed, and the search of the nearest points is performed within transverse ERP images (transverse equirectangular projection – the Cassini projection [5]). In equirectangular projection, a sphere is mapped onto a cylinder that is tangential to points on a sphere having the latitude equal to 0 degrees (Figure 23a). In transverse projection, the cylinder on which the sphere is mapped is rotated by 90 degrees, so it is tangential to points that have longitude equal to 0 degrees (Figure 23b). It changes the properties of the equirectangular projection in such a way, that the search for the nearest projected points can be performed only on the rows of the image.

|  |  |
| --- | --- |
| cylinders | cylinders |

Figure : Cylinders used in the projection of a sphere on a flat image in a) equirectangular projection and b) transverse equirectangular projection

A fast approximate reprojection of equirectangular image to transverse equirectangular image is used. In a first step, the length of all rows in an equirectangular image is changed to correspond to the circumference of the corresponding circle on a sphere (Figure 24a). In a second step, all columns of such image are expanded (Figure 24b), to be of the same length (Figure 24c).

transverse

Figure : Fast reprojection of an equirectangular image (a) to transverse equirectangular image (c). Black arrows show direction of change of size of respective rows and columns of images.

## Viewing space controller

A future version of TMIV 3 will include a viewing space controller.

When *viewing\_space\_present\_flag* is activated, the viewing space controller is in charge of applying to the viewport a smooth fade out to black according to an internal fading index computed in the decoder part in the viewing space controller (value 0 means no fade). This module computes this index from the viewport current position and orientation and from metadata related to the geometrical dimension of the viewing space and viewing direction constraints. The dimension of the viewing space is defined by the *primitive\_operation\_flag* through two operation alternatives which are either CSG (Constructed Solid Geometry) or interpolation. The interpolation mode makes use of metadata which lists in an ordered way the position and orientation of primitive cardinal shapes (cuboid, spheroid, half space). The CSG operation makes use of the elementary shapes which are themselves defined from primitive shapes either by CSG or interpolation. For all these modes, it is possible to compute a signed distance *SD(p)* which is zero at the frontier of the related shape, negative inside and positive outside, from which a positional fading index can be computed as follows:

positional fading index (p)= clamp((SD(p)+guard\_band\_size) / guard\_band\_size, 0, 1)

where *p* is the position of the viewport, and *guard\_band\_size* is the value of the signed distance from which the fading should start, and *clamp(a, min, max)* is the clamping function of a value a on the [min, max] interval. This first index should be combined multiplicatively by two orientational fading indexes related to the current viewport yaw and pitch respectively. For example, the orientation fading index for the yaw is computed as follows:

yaw fading index (p)= clamp((abs(yaw - primitive\_shape\_viewing\_direction\_yaw\_center)- primitive\_shape\_viewing\_direction\_yaw\_range + guard\_band\_orientation\_size) / guard\_band\_orientation\_size, 0, 1)

The viewing direction at a given position of the viewport is obtained from the set of individual values. In Figure 25, two modes of Viewing Space are illustrated, as well as viewing direction with the arrows.

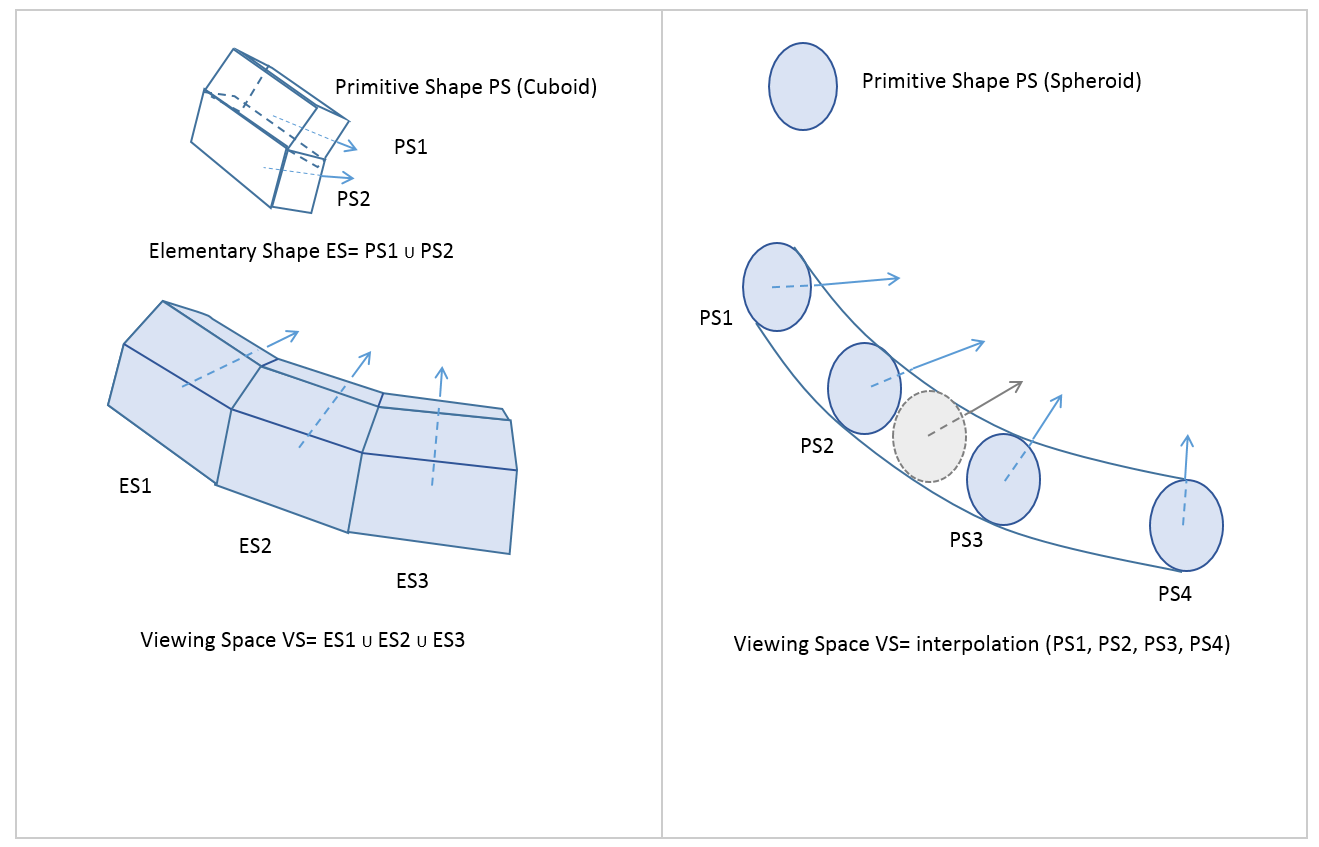


Figure : illustration of VS creation with additive CSG (left) and interpolation (right)

# Reference software

## Availability and use

The reference software (TMIV-SW) is publicly available on the Gitlab server at:

<https://gitlab.com/mpeg-i-visual/tmiv/>

The software is ISO C++17 conformant and does not require external libraries. Core experiments are expected to include the reference software as a subproject and introduce new components. Alternatively core experiments may fork the test model.

## Software architecture

Figure 26 provides a module dependency diagram (software architecture) of TMIV-SW. There is one library per component, two main executables (the encoder and decoder) and some components have a test executable. Catch2 is a header-only test framework[[4]](#footnote-4) that is not required but highly recommended. The Metadata library provides a full implementation of WD3 syntax and semantics. WD3 processes are spread across the Metadata, Renderer and AtlasDeconstructor libraries.

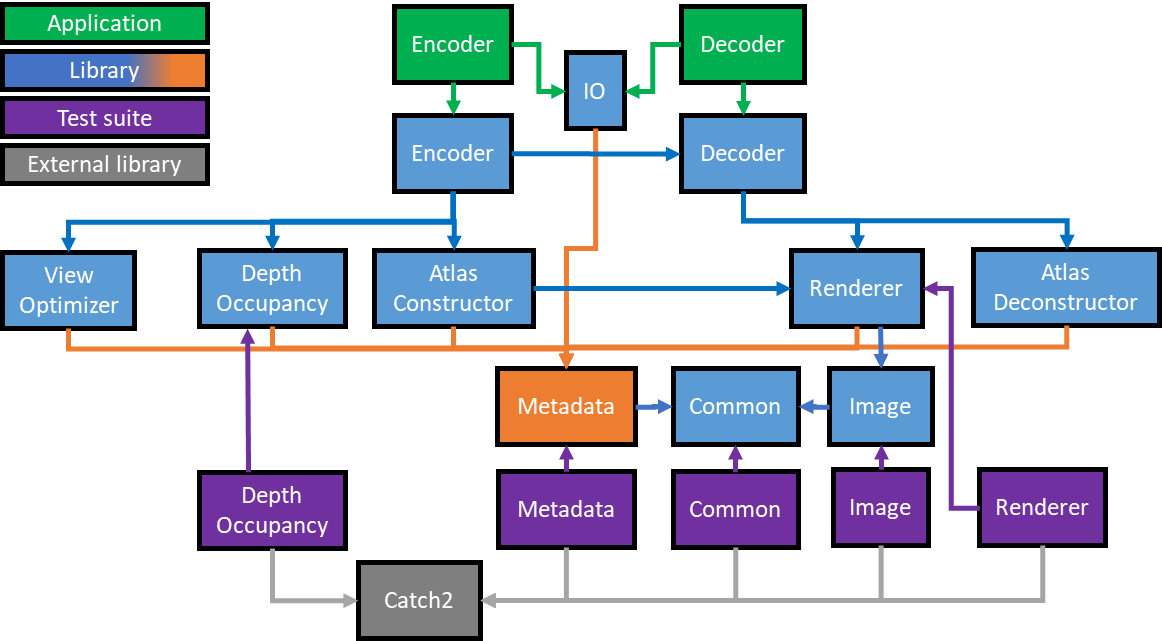


Figure : TMIV component dependency diagram. Arrows indicate "depends on" and "has access to". Dependencies are public. Hence when a library depends on the Metadata library, it implicitly depends on the Common library.

## Version strategy

TMIV uses the well-known semantic versioning strategy[[5]](#footnote-5). Version numbers are *x.y*[*.z*] with:

* *x* the major release (upped each MPEG meeting)
* *y* the minor release (adding functionality that does not impact the CTC)
* *z* the patch (for bugfixes)

Some of the functionality described in this document is not yet available in the x.0 release but is also not required for the CTC.

## Software coordination

In case of any related inquiries, please contact one of the software coordinators:

* Julien Fleureau, [julien.fleureau@technicolor.com](mailto:julien.fleureau@technicolor.com)
* Bart Kroon, [bart.kroon@philips.com](mailto:bart.kroon@philips.com)
* Bin Wang (王彬), [3130100819@zju.edu.cn](file:///E:\BS%20Sync\Projects\LFStreaming\Reports\MPEG\126%20Geneva%202019\TestModel\3130100819@zju.edu.cn)

Contributions should be in the form of git pull requests. Work of insufficient quality may be returned with comments. It is highly encouraged that proponents provide or extend a test suite. Proponents should be aware that only the IO library is allowed to perform file streaming (Because components should be composable.)

**References**

[1] *Call for Proposals on 3DoF+ Visual*, ISO/IEC JTC1/SC29/WG11 MPEG/N18145, Jan. 2019, Marrakesh, Morocco.

[2] J. Boyce, R. Doré, V. Kumar Malamal Vadakital (Eds.), “Working Draft 3 of Metadata for Immersive Media (Video)”, ISO/IEC JTC1/SC29/WG11 MPEG/N18794, October. 2019, Geneva, Switzerland.

[3] J. Jung, B. Kroon, J. Boyce, Common Test Conditions for Immersive Video, ISO/IEC JTC1/SC29/WG11 MPEG/N18789, October. 2019, Geneva, Switzerland.

[4] *Reference View Synthesizer (RVS) manual*, ISO/IEC JTC1/SC29/WG11 MPEG/N18068, Oct. 2018, Macao, China.

[5] J. Snyder, P. Voxland, “An album of map projections”, US Government Printing Office, Washington, 1989.

[6] J. Jylänki, “A thousand ways to pack the bin - a practical approach to two-dimensional rectangle bin packing”, 2010.

# Annex 1: TMIV configuration

The parameters setting the temporal aspect (i.e. frame parameters) of the TMIV-SW are set as follows:

*"startFrame": 0,*

*"numberOfFrames": 97,*

*"intraPeriod": 32,*

*“extraNumberOfFrames”: 203 (Optional)*

The *extraNumberOfFrames* option mirrors a decoded video a specified length. It is used for generating pose traces of 300 frame total.

*"maxEntities": 1,*

*"numGroups": 1,*

*"depthLowQualityFlag": false,*

*"OmafV1CompatibleFlag": true,*

When *maxEntities* >1 the TMIV encoder runs in object-based mode, otherwise it runs in regular mode.

*numGroups* defines the number of groups the input views will be distributed into where each group’s views can be encoded separately by the TMIV encoder.

*depthLowQualityFlag* indicates if the content has low quality depth maps and it is determined by the procedure illustrated in Annex 3.

*OmafV1CompatibleFlag* indicates if the produced bitstream is compatible with OMAF v1.

The source views and the associated parameters are set as follows:

*"SourceTexturePathFmt": "%s\_texture\_4096x2048\_yuv420p10le.yuv",*

*"SourceDepthPathFmt": "%s\_depth\_4096x2048\_yuv420p16le.yuv",*

*"SourceDirectory": ".",*

*"SourceCameraParameters": "ClassroomVideo.json",*

*"SourceCameraNames": [ "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14"]*

The *SourceCameraParameters* are in metadata format defined in section 3.3 and provide camera parameters and source resolutions per source view. There may be more cameras in this JSON, but the ones indicated by *SourceCameraNames* are used in the specified order.

The target view is set as follows:

*"OutputCameraName": ["v1"],*

*“PoseTracePath”: “Apt2.csv” (Optional, in this case OutputCameraName may be “viewport”)*

*"reconstruct": false,*

*"OutputDirectory": ".",*

Hereby *OutputCameraName* has to be a camera in the *SourceCameraParameters* file. The pose trace is optional and when provided it shall have the pose trace format defined in section 3.3. When *reconstruct* is true the decoder is being called from the encoder as well (useful for debugging purposes).

When the view optimizer and atlas constructor run as separate executables, then the paths for intermediate data and metadata files have to be specified:

*"BasicTexturePathFmt": “BAS\_SA\_R0\_Tt\_v%02d.yuv",*

*"BasicDepthPathFmt": "BAS\_SA\_R0\_Td\_v%02d.yuv",*

*"BasicMetadataPath": "BAS\_SA\_R0\_Tm\_vxx.bit",*

*"AdditionalTexturePathFmt": "ADD\_SA\_R0\_Tt\_v%02d.yuv",*

*"AdditionalDepthPathFmt": "ADD\_SA\_R0\_Td\_v%02d.yuv",*

*"AdditionalMetadataPath": "ADD\_SA\_R0\_Tm\_vxx.bit"*

The following fields specify the output of the encoder and the input of the decoder:

*"AtlasTexturePathFmt": "ATL\_SA\_R0\_Tt\_c%02d\_4096x3072\_yuv420p10le.yuv",*

*"AtlasDepthPathFmt": "ATL\_SA\_R0\_Td\_c%02d\_4096x3072\_yuv420p10le.yuv",*

*"AtlasMetadataPath": "ATL\_SA\_R0\_Tm\_c00.bit"*

When the atlas patch occupancy map generator and the renderer are run as separate executables, then the path format for the occupancy maps has to be specified:

*"AtlasPatchOccupancyMapFmt": "APO\_SA\_R0\_Td\_c%02d.yuv"*

The final paths to be specified are the output paths:

*"OutputTexturePath": "A17\_SA\_R0\_Tt\_v1\_4096x2048\_yuv420p10le.yuv",*

*"OutputDepthPath": "A17\_SA\_R0\_Td\_v1\_4096x2048\_yuv420p10le.yuv", (Optional)*

The config file selects one implementation per component, but the reference software may provide some alternatives (no view optimization, renderer / group-based renderer / multi-pass renderer, or no inpainting). Core experiments will add more implementations. A component is selected through *Method* parameters and method parameters are specified per component in a section with the name of the method. This section has to be present even when there are no method parameters (like in *ViewReducer* case below).

The configuration of components is hierarchical using the following pattern:

"Component": {

"SubcomponentMethod": "TheSubcomponentMethod"

"TheSubcomponentMethod": {

(...)

},

(...)

}

**Annex 2: Depth Low Quality Determination**

depthLowQualityFlag of a Boolean type (denoted as depth\_low\_quality\_flag in WD3 [2]) has been introduced within the TMIV config file to denote if the content has bad depth maps or not. However, it has been computed per MIV sequence included in the CTC [3] using the following preprocessing procedure:

* Few frames across all views per sequence are synthesized directly from the source views with source depth maps using the TMIV synthesizer (i.e. single pass with all views made available at its input but the one being synthesized) & PSNR values are computed.
* A unified threshold across all sequences is chosen such that it produces depth\_low\_quality\_flag that reflects the best merging-mode / synthesis results recommended for all sequences.
* The flag determination needs to be revisited every time depth maps or synthesis tools get updated.

For the current MIV content, a threshold is set to be 29dB resulting in depthLowQualityFlag being true for the IntelFrog (E) case and false for other sequences. The average PSNR results computed by synthesizing views of the first frame of the sequences are shown in the table below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Seq. | A | B | C | J | D | E | L |
| Y-PSNR | 33.747 | 34.413 | 39.449 | 31.094 | 37.001 | 27.91 | 29.89 |
| depthLowQualityFlag | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

[Ed.(BK): Automate this in the next version of TMIV.]

1. Please note that the Atlas Patch Occupancy Map Generator is called AtlasDeconstructor in TMIV-SW. The component has the additional (optional) functionality of reconstructing pruned views. [↑](#footnote-ref-1)
2. <http://json.org/> [↑](#footnote-ref-2)
3. <https://mpeg.chiariglione.org/standards/mpeg-i/omnidirectional-media-format> [↑](#footnote-ref-3)
4. <https://github.com/catchorg/Catch2> [↑](#footnote-ref-4)
5. <https://semver.org/> [↑](#footnote-ref-5)