**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 MPEG2019/m52414**

**January 2020, Brussels, Belgium**

|  |
| --- |
| **Source Interdigital** |
| **Status Input contribution** |
| **Title CE2.6.2: Graph-Based Pruning for Natural Contents** |
| **Author Julien Fleureau, Renaud Doré, Franck Thudor, Thierry Tapie, Gérard Briand, Bertrand Chupeau** |
|  |

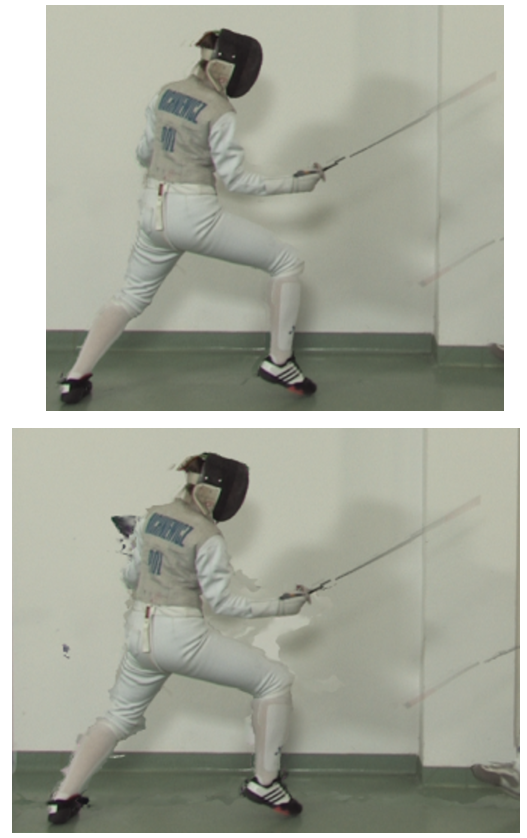
# Context

Natural contents are very different from Computer-Generated ones because of the quality of their depth maps. In addition to be less accurate, these former depth maps also suffer from inter-view inconsistency because, most of the time, the depth estimation is strongly based on photo-consistency between groups of close cameras. It is especially the case of IntelFrog (E) and PoznanFencing (L) contents where the depth maps were originally computed making use of multiple stereo estimations. As a consequence, two close cameras tend to be more consistent in terms of geometry information and it is even more true when considering large rigs of cameras.

At the synthesis stage, directly mixing the information coming from inconsistent cameras creates important artifacts (see Figures 1 and 2 - bottom) which are not acceptable for a proper visualization. A very classical and efficient way to mitigate this issue consists in weighting the contribution of each input cameras depending on their “distance” to the viewport to render. The “distance” may be for instance derived from the Euclidean distance of the viewpoint to synthesize to the input camera centers or from any other weighting function. Doing that way clearly improves the quality of the final rendering (see Figures 1 and 2 – top) because the synthesis process mainly uses the “local” consistent information.



**Figure 1: Synthesis with (top) and without (bottom) view weighting for PoznanFencing.**



**Figure 2: Synthesis with (top) and without (bottom) view weighting for PoznanFencing (zoom).**

# Pruning-Aware Synthesis

The cornerstone of the MIV process to transmit MVD contents relies on the concept of pruning. The idea is clearly to remove the redundancy that may exist between the input views to reduce the amount of data to transmit to the decoder.

To do so, the current MIV technology relies on a set of patches to convey the original MVD scene. Each patch is a rectangular sub-part of an input view and the whole set of patches is designed (at the pruning stage) to be non-redundant but complementary in order to convey the full input volumetric scene. The set of patches is packed into atlases and in addition to the input views parameters, the current MIV metadata also embeds patch packing and mapping information to be able to recover the set of input pruned views (see Figure 3).



**Figure 3: Example of pruned view for IntelFrog.**

Let’s now consider again the synthesis stage making use of these pruned views. As mentioned in the previous part, an important concept to enhance the quality of view interpolation of Natural Contents is to make use of view weighting to mitigate the inconsistency of the input depths. However, once pruned, the weighting strategy may be not achievable anymore. Indeed, a source camera which may be very useful to recover the viewport because its associated weight is high may have been pruned. At the decoding stage, it is not possible anymore to access to some important information to recover the current view (because of the pruning) and it may lead to important visual artifacts (similar to the ones observable in Figures 1 and 2 – bottom).

If, at the decoding stage, one would have access to the pruning information (i.e. the pruning relations between each camera as decided at the encoding stage), one would be able to recover the pruned parts of each input view. One could therefore properly apply the view weighting strategy and therefore enhance the quality of the final rendering.

# Graph-Based Pruning Recovery

Transmitting the pruning information is however a bit tricky because within a given pruned view, a given pixel may have been pruned because it was found in any of the input cameras. At first sight, transmitting the pruning information would therefore come down to the transmission of a kind of look-up table pairing all pruned pixels of a first view with its “representative” (i.e. the pixel which has not been pruned and which is considered as similar to the current pruned one) of another view. Moreover, this table should be updated at frame rate because the pruning is done independently for each temporal frame.

Of course, this way to do is far too expensive because the amount of data would be huge (at least one pairing per pruned pixel per temporal frame).

As an alternative, we propose an indirect approach to achieve the same goal. Instead of explicitly transmitting the pruning information, we propose to constrain the pruning process to make it reversible at the price of very few additional extra metadata. Doing this way, one just has to apply the reverse process to “unprune” the pruned views, which is indeed equivalent to the transmission of the set of pruning information.

## Additional Metadata

The set of additional metadata to transmit relies on the concept of pruning graph which is a bit more general than a simple pruning order and which can be viewed as a generalization of a pruning hierarchy (see TMIV 3.0).

A pruning graph is a regular directed graph constructed at the encoding / pruning stage where each node corresponds to a view to prune. It is defined as follows:

* A node *Ni* of the pruning graph is connected to a node *Nj* if the view associated to *N*j should be **firstly** pruned with respect to the view associated to node *Ni* at the pruning stage.
* Each basic view is associated to a root node of this graph as it should not be pruned at all.

Transmitting the pruning graph comes down to transferring, once per sequence, for each view, the node ids of its children or direct parents in the pruning graph. A first possible way to do could be to add in the MIV metadata set a new list containing, for each camera, the list of children ids (see Table 1). If the number of cameras is small (typically less than 64), then a mask / bit array may be used alternatively to describe the pruning relation, each ith bit being set to 1 if the ith node is a child of the node associated to the current camera (see Table 2).

|  |  |
| --- | --- |
| pruning\_graph( ) { | **Descriptor** |
| **num\_cameras\_minus1** | u(16) |
| for ( i= 0; i <= num\_cameras\_minus1; i++) { |  |
| **num\_children\_minus1**[ i ] | u(16) |
| for ( j= 0; j <= num\_children\_minus1[i]; j++) |  |
| **children\_id**[ i ][j] | u(16) |
| } |  |

**Table 1: Proposed additional metadata to convey pruning graph (array version).**

|  |  |
| --- | --- |
| pruning\_graph ( ) { | **Descriptor** |
| **num\_cameras\_minus1** | u(16) |
| for ( i= 0; i <= num\_cameras\_minus1; i++) |  |
| **children\_id\_mask**[i] | u(64) |

**Table 2: Proposed additional metadata to convey pruning graph (mask version).**

The pruning relation could be also directly integrated within the camera parameters list (either as an array or as a mask) as a new parameter for each camera (see Tables 3 and 4).

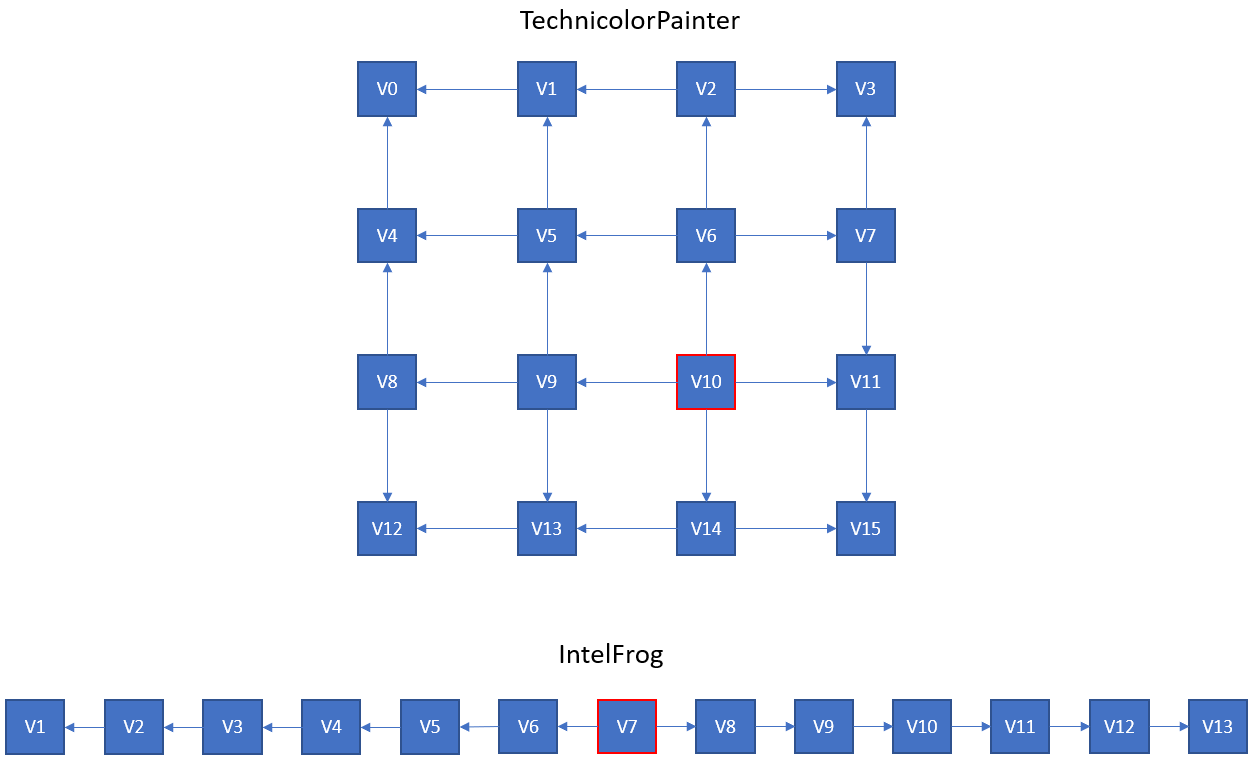
|  |  |
| --- | --- |
| camera\_params\_list( ) { | **Descriptor** |
| **num\_cameras\_minus1** | u(16) |
| for ( i= 0; i <= num\_cameras\_minus1; i++) { |  |
| **cam\_pos\_x**[ i ] | u(32) |
| **cam\_pos\_y**[ i ] | u(32) |
| **cam\_pos\_z**[ i ] | u(32) |
| **cam\_yaw**[i ] | u(32) |
| **cam\_pitch**[ i ] | u(32) |
| **cam\_roll**[ i ] | u(32) |
| } |  |
| **intrinsic\_params\_equal\_flag** | u(1) |
| for ( i = 0; i <= intrinsic\_params\_equal\_flag ? 0 : num\_ cameras\_minus1; i++ ) |  |
| camera\_intrinsics( [ i ] ) |  |
| **depth\_quantization\_params\_equal\_flag** | u(1) |
| for ( i = 0; i <= depth\_quantization\_equal\_flag ? 0 :  num\_cameras\_minus1; i++ ) |  |
| depth\_quantization( [ i ] ) |  |
| **pruning\_params\_equal\_flag** | u(1) |
| for ( i= 0; i <= pruning\_params\_equal\_flag ? 0 : num\_cameras\_minus1; i++) { |  |
| **num\_children\_minus1**[ i ] | u(16) |
| for ( j= 0; j <= num\_precedences\_minus1[i]; j++) |  |
| **children\_id**[ i ][j] | u(16) |
| } |  |

**Table 4: Proposed embedded additional metadata to convey pruning graph (array version).**

|  |  |
| --- | --- |
| camera\_params\_list( ) { | **Descriptor** |
| **num\_cameras\_minus1** | u(16) |
| for ( i= 0; i <= num\_cameras\_minus1; i++) { |  |
| **cam\_pos\_x**[ i ] | u(32) |
| **cam\_pos\_y**[ i ] | u(32) |
| **cam\_pos\_z**[ i ] | u(32) |
| **cam\_yaw**[i ] | u(32) |
| **cam\_pitch**[ i ] | u(32) |
| **cam\_roll**[ i ] | u(32) |
| } |  |
| **intrinsic\_params\_equal\_flag** | u(1) |
| for ( i = 0; i <= intrinsic\_params\_equal\_flag ? 0 : num\_ cameras\_minus1; i++ ) |  |
| camera\_intrinsics( [ i ] ) |  |
| **depth\_quantization\_params\_equal\_flag** | u(1) |
| for ( i = 0; i <= depth\_quantization\_equal\_flag ? 0 :  num\_cameras\_minus1; i++ ) |  |
| depth\_quantization( [ i ] ) |  |
| **pruning\_params\_equal\_flag** | u(1) |
| for ( i= 0; i <= pruning\_params\_equal\_flag ? 0 : num\_cameras\_minus1; i++) |  |
| **children\_id\_mask**[i] | u(64) |
| } |  |

**Table 5: Proposed embedded additional metadata to convey pruning graph (mask version).**

Such an additional metadata therefore describes the pruning relations in a quite generic and lightweight manner. Additionally, some examples of pruning graph are given in Figure 4 for TechnicolorPainter and IntelFrog.



**Figure 4: Example of pruning graphs for TechnicolorPainter (top) and IntelFrog (bottom).**

Note that this metadata is compatible with the notion of group Id introduced at MPEG128. When the notion of group is activated (num\_group\_minus1 > 0), the elementary links described by previous metadata should create as many graphs as there are groups.

We now show how a quite generic reversible pruning procedure can be elaborated on the only basis of this additional metadata knowledge.

## Encoding stage

At the encoding stage, the pruning process is performed. To do so, a pruning graph is decided, and the pruning procedure is then launched from the root node to the leaves.

To ensure the reversibility of the pruning procedure, the pruning graph as well as the pruning process should be constrained. We propose here one way to achieve this objective.

Regarding the graph structure itself, the only constraint to ensure is to make the pruning graph acyclic so that a given view is pruned once all its direct parents and indirect parents (grandparents, great-grandparents, …) have been “fully” pruned. The pruning graph should be therefore browsable from the root(s) to leaves without cycle.

Regarding the process itself, two simple rules must be respected:

1. A pixel *p* of a view to prune should be pruned if and only if all reprojections in its direct parent views are “similar” to itself. The notion of similarity is implementation dependent and could be for instance based on a geometric and / or color criterion.
2. If one of its parent views is already pruned at the reprojected position or if the reprojection does not reproject into the parent viewport, then the previous rule should be extended recursively to grandparents.

Ensuring the pruning with respects to all parents ensures the reversibility of the procedure.

Recursively pruning with respect to grandparents in case of an early pruning ensures that no drift occurs when the similarity is propagated towards the root node(s). It is indeed necessary because the similarity function is generally not transitive as it often relies on a thresholded squared distance.

The way the pruning graph is constructed is of course implementation dependent as long as the set of previous rules is followed. However, a good practice to avoid too much redundancy is to “connect” the graph nodes depending on the “proximity” of their associated viewport.

At the end of the day, respecting the previous rules during pruning implicitly ensures that a deterministic procedure exists at the decoding stage to reverse this pruning stage (see next section).

## Decoding stage

* + 1. Unpruning input views

At the decoding stage, if the following previous pruning rules have been followed, the pruning graph may now be used again to “unprune” the pruned input views. To do so, another simple “reverse” procedure may be applied to recover from the pruning process. It is summed up hereafter:

1. A pruned pixel *p* is “unpruned” making use of any unpruned pixel from its parent which reprojection reprojects at *p*.
2. If no unpruned pixel of parent views reprojects at *p*, then the previous rule should be extended recursively to grandparents.

Doing so makes possible to “unprune” the pruned views and then apply the weighting strategy at the synthesis stage properly.

* + 1. Weight recovery

“Unpruning” all source views before synthesizing the viewport of interest may be however a bit overkill and a smarter / faster approach can be advantageously envisioned to achieve the same goal, still using the pruning graph.

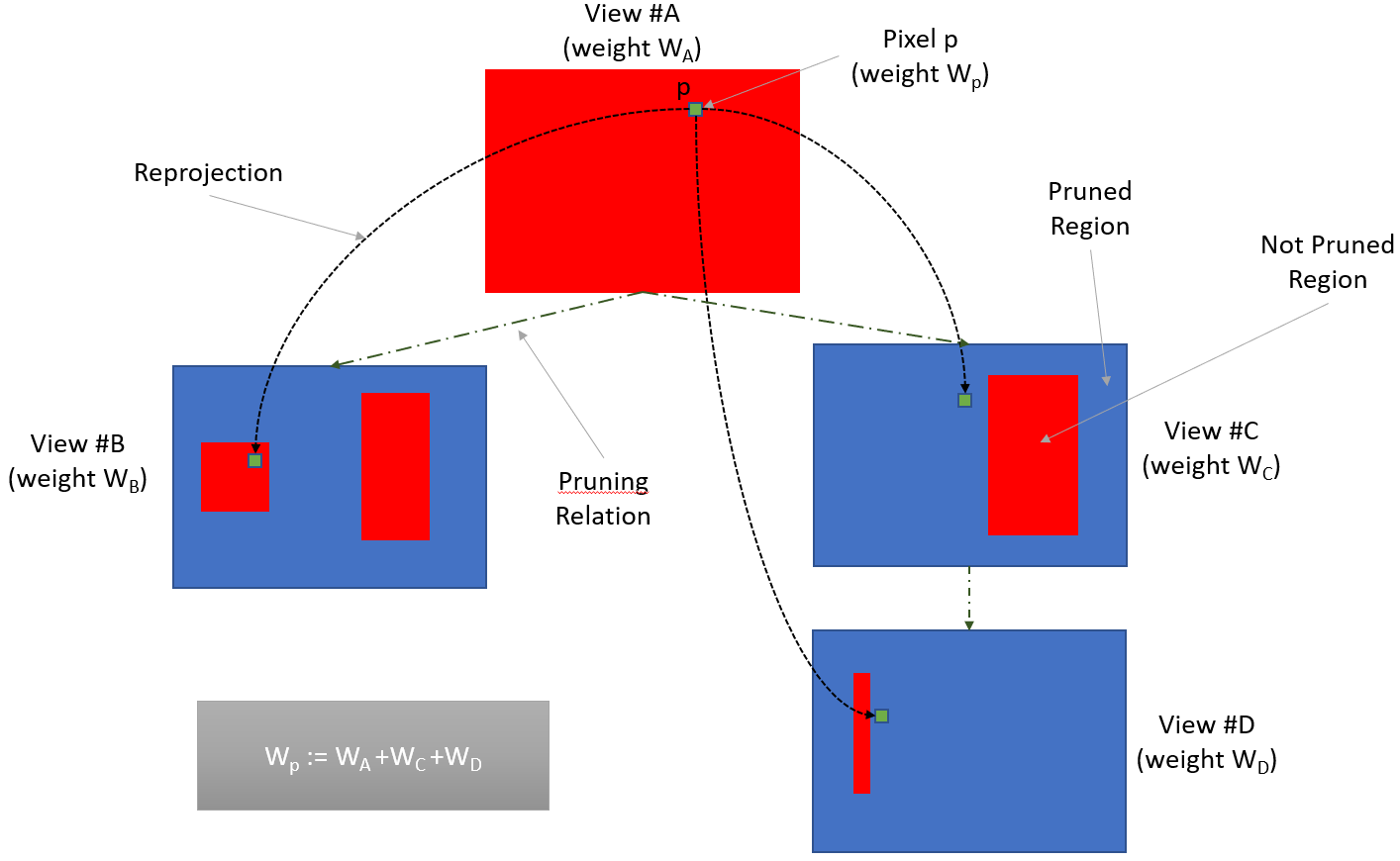
Instead of making use of “unpruned” source views to apply the view weighting strategy, this alternative approach simply updates the weight of each non-pruned pixels at the synthesis stage to take into account that they could “represent” other pruned pixels in the descendant hierarchy of the pruning graph (see Figure 5).

Doing so allows for directly computing / recovering the good weight to use at the synthesis stage for a given non-pruned pixel without computing explicitly the unpruned input views.

To correctly assess the weight of a non-pruned pixel at the synthesis stage, the following procedure can be applied. Let’s consider again a non-pruned pixel *p* of a view associated to a node *N* of the pruning graph. Let’s call *wP = wN* its initial weight (which only depends on the “distance” of the view it belongs to, to the view to synthesize). Then this weight is updated as follows:

1. If the pixel *p* reprojects into one of its child views at a pruned pixel then its weight is cumulated with the weight *wO* of this “child” view by *wP* := *wP* + *wO* and the process is recursively repeated to the grandchildren.
2. If the pixel *p* does not reproject into one of its child viewports, then the previous rule is extended recursively to the grandchildren.
3. If the pixel *p* reprojects into one of its child views at an unpruned pixel then its weight is let unchanged and no more inspection of the graph is performed toward grandchildren.

This latter procedure is of course only valid if the pruning process has been done according to what has been described in section 3.2.



**Figure 5: Weight recovery procedure**

## Implementation details

Note that the proposed implementation of CE2 only handles one single basic camera for sake of simplicity. As a consequence, a simple IDCCViewOptimizer has been introduced to ensure the respect of this condition. To that end, the IDCCViewOptimizer simply returns the closest view to the viewport camera (as defined in each JSON file of each content) as the basic view and consider all the remaining ones as additional views.

Moreover, for this CE, the similarity measure used for the pruning stage (IDCCPruner module) is a simple geometric-only criterion which compares the relative depth of two collocated pixels within the reprojection frame to decide is they are similar or not. No color criterion is involved.

The group-based feature has been inactivated at the encoding and decoding stage. The IDCC synthesizer (cf. CE2.6.1 – m52413) was used for decoding purposes because the weight recovery strategy was easier to implement within this code. However, this latter feature is conceptually synthesizer agnostic.

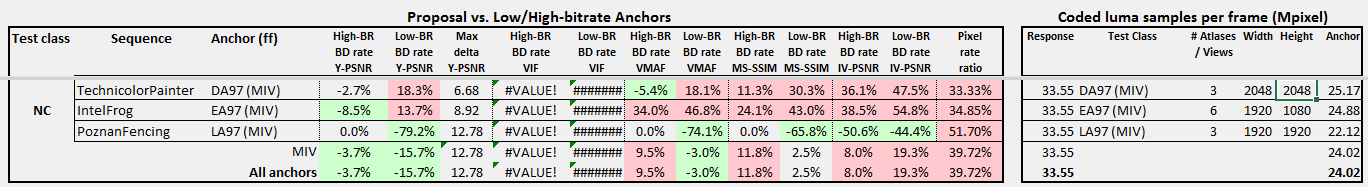
Finally, a generic method of pruning graph construction is proposed / implemented in the provided code which makes use of simple graph algorithms (minimum bottleneck spanning trees especially). It allows for the computation of a pruning graph whatever the geometry (examples given in Figure 4 have been obtained making use of this procedure). It is of course not the only way to obtain such graphs and other algorithms (including the one of TMIV3.0) may be used instead.

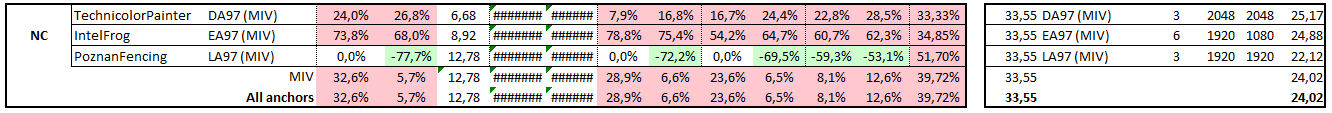
# Results

Overall results are provided in Table 6 (for comparison with TMIV3.0) and 7 (for comparison with CE2.6.1), and Figures 6, 7 and 8 show the rate / distortion diagrams for TechnicolorPainter, IntelFrog and PoznanFencing.

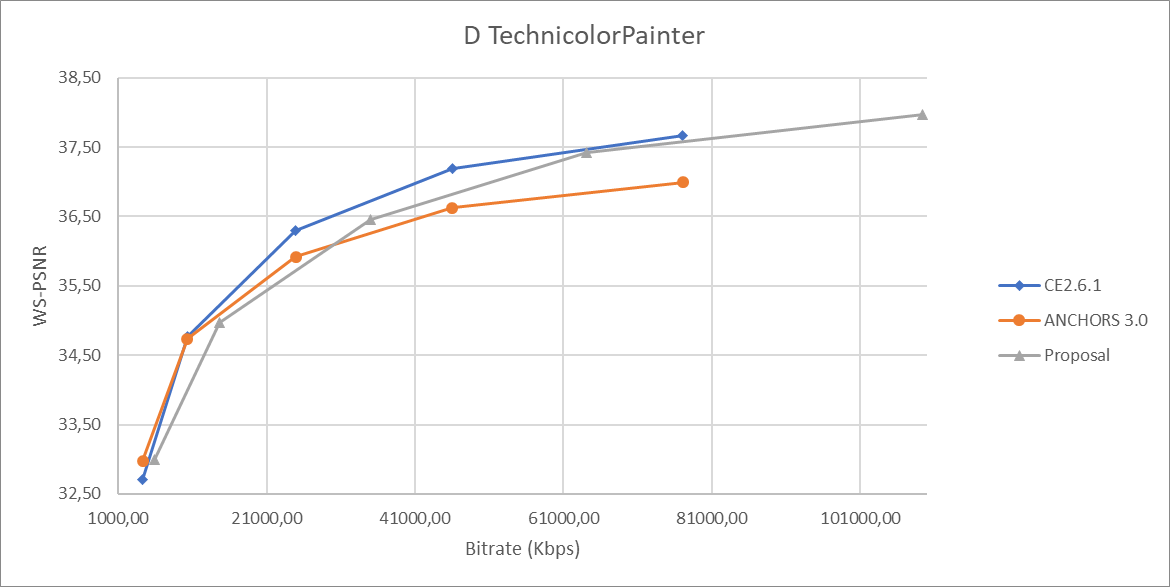
Note that a single 4K x 4K atlas has been used on purpose for the packing of the patches for all contents, and to stick to the anchor pixel rate. We did not try to optimize the pixel rate further as this resolution was considered as a low pixelrate by the group and as such a framesize can be realtime decoded in most devices included smartphones. We also want to highlight here that a notion of profile including typical atlas sizes should be discussed at a certain point by the group.

The best improvement with Y-PSNR is obtained for PoznanFencing but a good behavior of the proposed approach can also be observed on TechnicolorPainter and IntelFrog for the 3 highest QPs. CE2.6.1 quantitatively outperforms the proposal (see Table 7) for TechnicolorPainter and IntelFrog but subjective results tend to mitigate (and even invalidate for IntelFrog) this apparent gain as explained in the following.

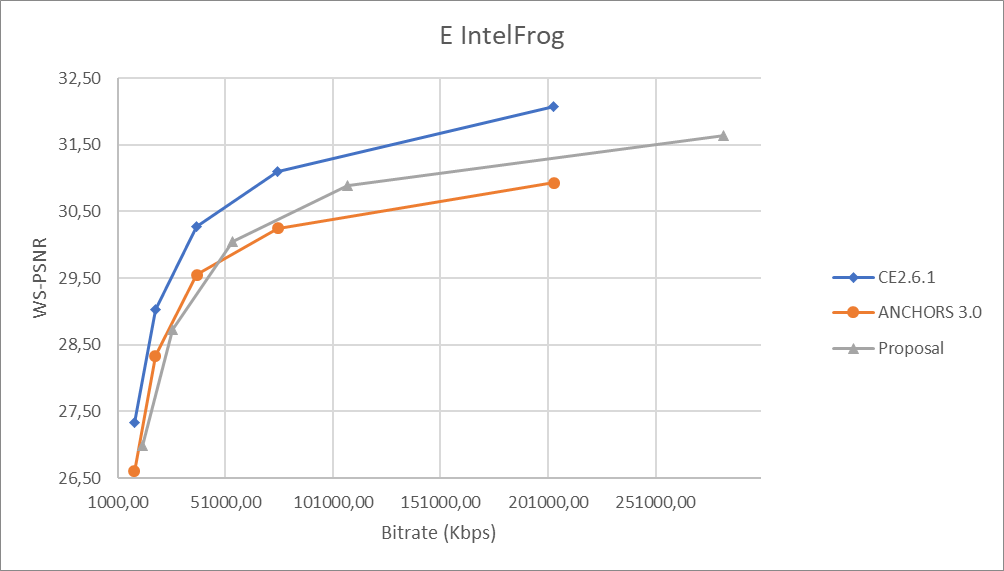
**Table 6: Overall results (TMIV3.0 vs proposal)**

****

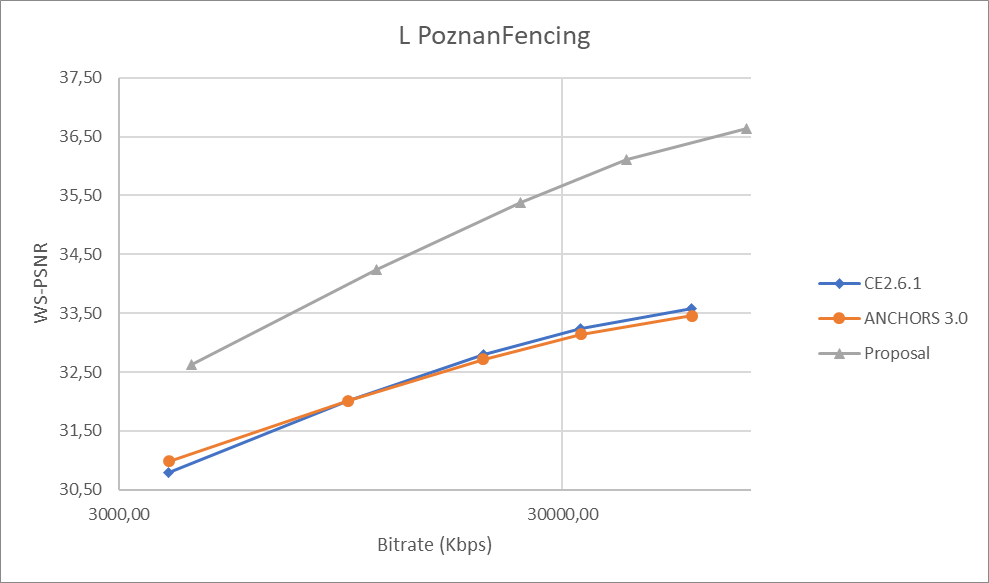
**Table 7: Overall results (CE2.6.1 vs proposal)**



**Figure 6: Rate / Distortion diagram for TechnicolorPainter.**

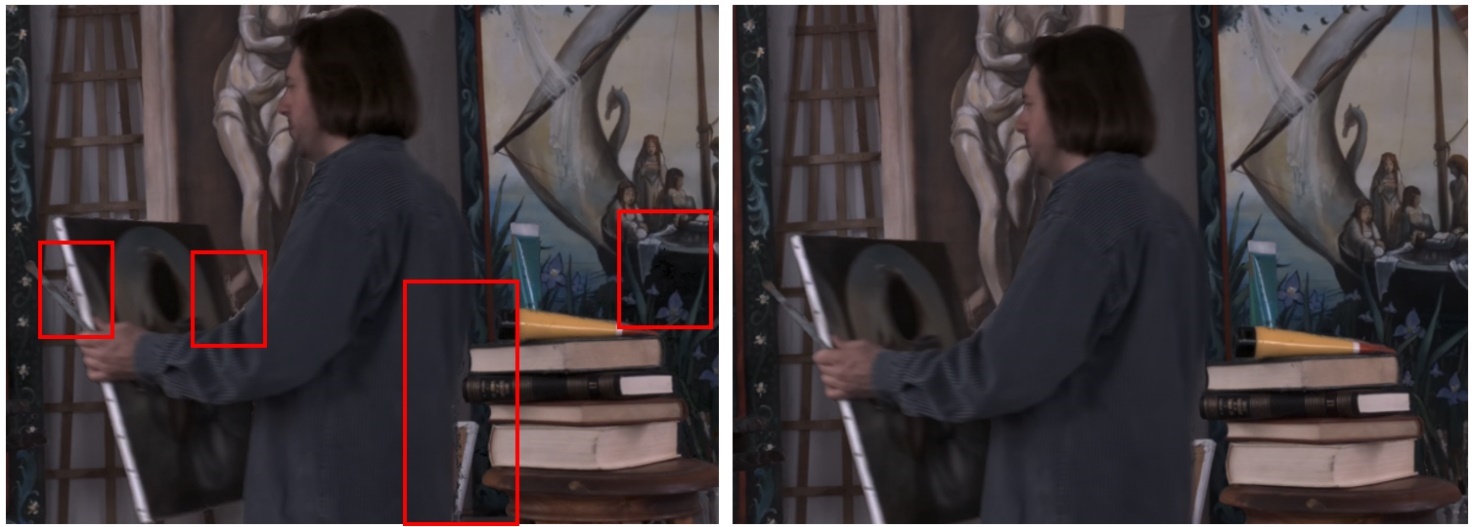


**Figure 7: Rate / Distortion diagram for IntelFrog.**



**Figure 8: Rate / Distortion diagram for PoznanFencing.**

Indeed, a subjective comparison has also been performed and proposed in associated side-by side videos. It tends to show slightly better performances on D (see Figure 9), comparable performances on E but better performances on L (see Figure 1 and 2 – top) when comparing with TMIV3.0. These results are quite logical if we consider the several improvements / refinements of the depth maps of D and E which have been performed since the beginning of the MIV standard. PoznanFencing is still the “worst” content in terms of depth quality / consistency where the additional proposed metadata is meaningful.



**Figure 9: TechnicolorPainter TMIV3.0 (left) and proposal (right).**

When comparing with CE2.6.1, same conclusions can be drawn regarding D (slight improvement) and L (clear improvement), but E is even significantly improved because of the absence of any missing content (see Figure 10) and this, despite the lower quantitative performances of our approach versus CE2.6.1 (see Figure 7). The differences may be partly due to the fact that using the IDCC Synthesizer (CE2.6.1) in combination with the group-based rendering seems to be suboptimal in this case which could be possibly and advantageously mitigated with the proposed approach.



**Figure 10: IntelFrog CE2.6.1 (left) and proposal (right).**

# Recommendations

We recommend embedding the pruning information in the metadata to be able to properly handle and render inconsistent MVD inputs. One possible implementation is proposed in this contribution. Quite important use cases may be addressed with such an additional metadata including sports event broadcasting where large rigs of cameras are often used and where the depth consistency is hard to ensure real-time. It is compatible with the notion of group Id.

We recommend addressing the notion of profile to define typical atlas resolutions.

We recommend questioning the validity of objective metrics when inconsistent with subjective quality.

# References

1. J. Jung, B. Kroon, J. Boyce, Common Test Conditions for Immersive Video, ISO/IEC JTC1/SC29/WG11 MPEG/N18789, October. 2019, Geneva, Switzerland.
2. A. Dziembowski, “Description of Immersive Video Core Experiments 2: Pixel Pruning”, ISO/IEC JTC1/SC29/WG11, N18933, October 2019, Geneva, Switzerland.
3. J. Fleureau, R. Doré, F. Thudor, T. Tapie, G. Briand, B. Chupeau, “CE2.6.1: Synthesizer”, ISO/IEC JTC1/SC29/WG11, M52414, January 2020, Brussels, Belgium.