

---

---

**Information technology — Coding of  
audio-visual objects —**

Part 16:

**Animation Framework eXtension (AFX)**

**AMENDMENT 2: Frame-based Animated  
Mesh Compression (FAMC)**

*Technologies de l'information — Codage des objets audiovisuels —*

*Partie 16: Extension du cadre d'animation (AFX)*

*AMENDEMENT 2: Compression trame par trame de maillage animé  
(FAMC)*

**PDF disclaimer**

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-16:2006/AMD2:2009



**COPYRIGHT PROTECTED DOCUMENT**

© ISO/IEC 2009

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office  
Case postale 56 • CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail [copyright@iso.org](mailto:copyright@iso.org)  
Web [www.iso.org](http://www.iso.org)

Published in Switzerland

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 2 to ISO/IEC 14496-16:2006 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 14496-16 introduced several animation models as methods of deforming a mesh. Amendment 2 to ISO/IEC 14496-16:2006 deals with decoding animation data (mainly vertex coordinates and attributes, temporally updated) independently of a mesh deformation model.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-16:2006/AMD2:2009

# Information technology — Coding of audio-visual objects —

Part 16:

## Animation Framework eXtension (AFX)

### AMENDMENT 2: Frame-based Animated Mesh Compression (FAMC)

After 5.9, add the following new subclause:

#### 5.10 Frame-based Animated Mesh Compression (FAMC) stream

##### 5.10.1 Overview

FAMC is a tool to compress an animated mesh by encoding on a time basis the attributes (position, normals ...) of vertices composing the mesh. FAMC is independent on the manner how animation is obtained (deformation or rigid motion). The data in a FAMC stream is structured in segments of several frames. Each segment can be decoded individually. Within a segment, a temporal prediction model, called *skinning*, is represented. The model is used for motion compensation inside the segment. The FAMC bitstream structure is illustrated in Figure AMD2.1.

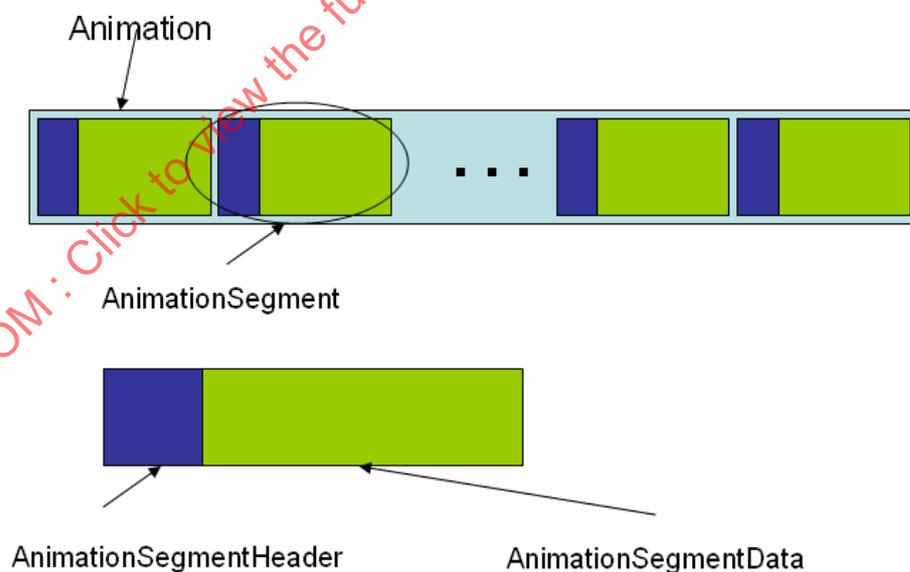


Figure AMD2.1 — FAMC bitstream structure.

Each decoded animation frame updates the geometry and possibly the attributes (or only the attributes) of the 3D graphic object that FAMC is referred to.

An animation segment contains two types of information:

- A header buffer indicating general information about the animation segment (number of frames, attributes to be updated...).
- A data buffer containing:
  - The skinning model used for 3D motion compensation consists in a segmentation of the 3D mesh into clusters and is specified by:
    - the **partition** information, i.e. the segmentation of the 3D object vertices into clusters,
    - a set of **animation weights** connecting each vertex of the 3D object to each cluster and
    - the motion data described in terms of a 3D **affine transform** for each cluster and for each animation frame.
  - The **residual errors** per vertex equal with the difference between the real value and the one predicted by the skinned motion compensation model, that are encoded with one of the following combination
    - a Discrete Cosine Transform performed on the entire animation segment (referred in this document as DCT)
    - an integer-to-integer Wavelet Transform performed on the entire animation segment (referred in this document as Lift).
    - Layer based decomposition (referred in this document as LD)
    - DCT followed by LD
    - Lift followed by LD

The prediction residual errors may correspond to geometric and/or attribute data.

Figure AMD2.2 illustrates the FAMC decoding process.

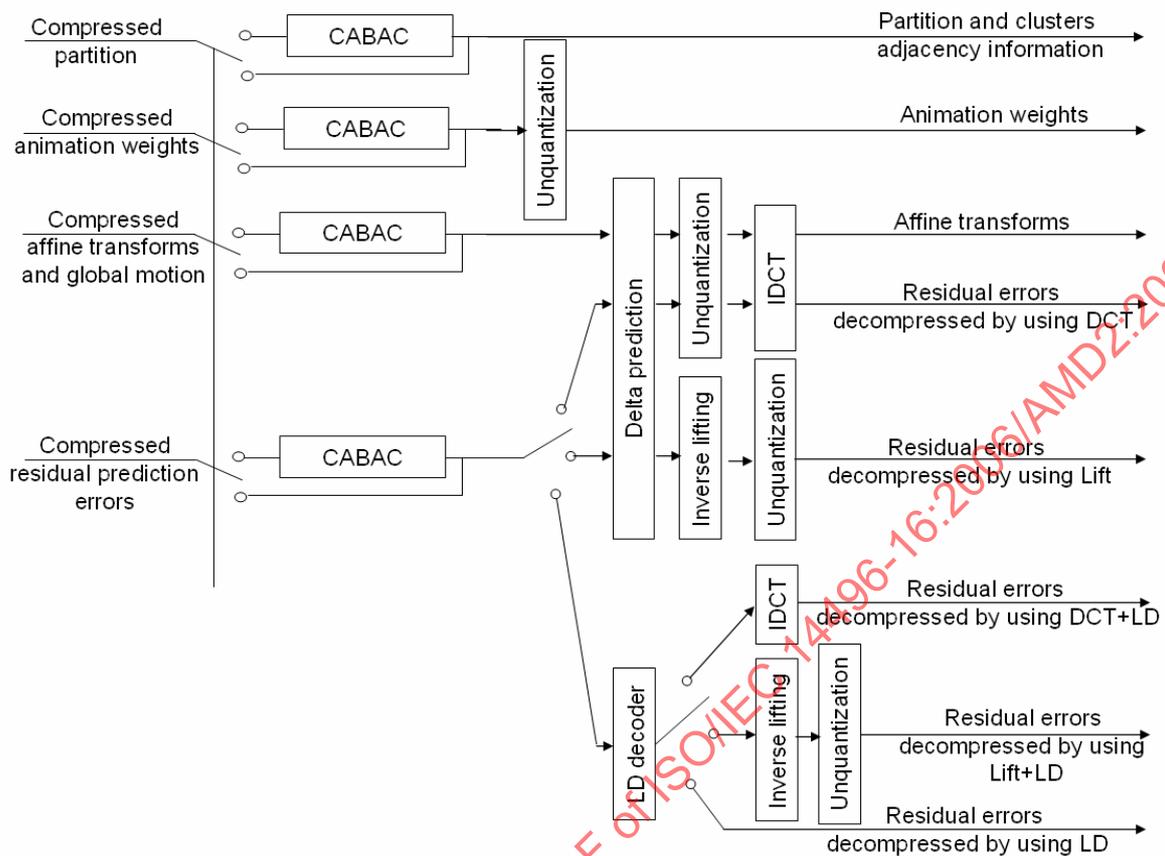


Figure AMD2.2 — FAMC decoding process.

The following sections describe in detail the structure of the FAMC stream.

### 5.10.2 FAMC inclusion in the scene graph

FAMC is associated with an IndexedFaceSet by using the BitWrapper mechanism with value of field *type* equals to 2.

### 5.10.3 FAMC class

#### 5.10.3.1 Syntax

```
class FAMCAnimation{
do{
    FAMCAnimationSegment animationSegment;
    bit(32) * next;
}
while (next==FAMCAnimationSegmentStartCode);
}
```

#### 5.10.3.2 Semantics

**FAMCAnimationSegmentStartCode:** a constant that indicates the beginning of a FAMC animation segment.

FAMCAnimationSegmentStartCode = 00 00 01 F0.

**5.10.4 FAMCAAnimationSegment class**

**5.10.4.1 Syntax**

```
class FAMCAAnimationSegment {
    FAMCAAnimationSegmentHeader header;
    FAMCAAnimationSegmentData data;
}
```

**5.10.4.2 Semantics**

**FAMCAAnimationSegmentHeader:** contains the header buffer.

**FAMCAAnimationSegmentData:** contains the data buffer.

**5.10.5 FAMCAAnimationSegmentHeader class**

**5.10.5.1 Syntax**

```
class FAMCAAnimationSegmentHeader {
    unsigned int (32) startCode;
    unsigned int (8) staticMeshDecodingType
    unsigned int (32) animationSegmentSize
    bit(4) animatedFields;
    bit(3) transformType;
    bit(1) interpolationNeeded;
    bit(2) normalsPredictionStrategy;
    bit(2) colorsPredictionStrategy;
    bit(4) otherAttributesPredictionStrategy;
    unsigned int (32) numberOfFrames;
    for(int f = 0; f < numberOfFrames; f++) {
        unsigned int (32) timeFrame[f];
    }
}
```

**5.10.5.2 Semantics**

**startCode:** a 32-bit unsigned integer equals to **FAMCAAnimationSegmentStartCode**.

**staticMeshDecodingType:** a 8-bit unsigned integer indicating if the static mesh is encoded within the FAMC stream and which decoder should be used. Table AMD2.1 summarizes all possible configurations.

**Table AMD2.1 —First frame decoding type: all possible configurations.**

<b>firstFrameDecodingType value</b>	<b>First frame decoding type</b>
0	The first frame is not encoded within the FAMC stream and should be read directly from the BIFS stream.
1-7	Reserved for ISO purposes

**animationSegmentSize:** a 32-bit unsigned integer describing the size in bytes of the current animation segment.

**animatedFields:** a 4-bit mask indicating which fields are animated. Table AMD2.2 summarizes all possible configurations.

Table AMD2.2 — Animated fields: all possible configurations.

	B1	B2	B3	B4
0	Coordinates animated	Normals animated	Colors animated	Other attributes animated
1	Coordinates not animated	Normals not animated	Colors not animated	Other attributes not animated

**transformType:** a 3-bit mask indicating the transform used for encoding the prediction residual errors. Table AMD2.3 summarizes all possible configurations.

Table AMD2.3 — Transform type: all possible configurations.

transformType value	Method used
0	Lift
1	DCT
2	LD
3	Lift + LD
4	DCT+ LD
5	Reserved for ISO purposes
6	Reserved for ISO purposes
7	Reserved for ISO purposes

**numberOfFrames:** a 32-bit unsigned integer indicating the number of frames to be decoded in the current animation segment.

**interpolationNeeded:** one bit indicating if, after decoding, animation frames have to be interpolated. If zero, all the animation frames are obtained from direct decoding.

**normalsPredictionStrategy:** a 2-bit mask indicating the prediction strategy for normals. Table AMD2.4 summarizes all possible configurations.

Table AMD2.4 — Normals prediction strategy: all possible configurations.

normalsPredictionStrategy value	Prediction used
0	Delta
1	Skinning
2	Tangential skinning
3	Adaptive

Note: the prediction is computed with respect to the reference static mesh as defined in the scene graph.

**colorsPredictionStrategy:** a 2-bit mask indicating the prediction strategy for colors. Table AMD2.5 summarizes all possible configurations.

**Table AMD2.5 — Color prediction strategy: all possible configurations.**

colorsPredictionStrategy value	Prediction used
0	Delta
1	Reserved for ISO purposes
2	Reserved for ISO purposes
3	Reserved for ISO purposes

Note: the prediction is computed with respect to the reference static mesh as defined in the scene graph.

**otherAttributesPredictionStrategy:** a 4-bit mask indicating the prediction strategy for other attributes. Table AMD2.6 summarizes all possible configurations.

**Table AMD2.6 — Other attributes prediction strategy: all possible configurations.**

otherAttributesPredictionStrategy value	Prediction used
0	Delta
1	Reserved for ISO purposes
2	Reserved for ISO purposes
3	Reserved for ISO purposes

NOTE the prediction is computed with respect to the reference static mesh as defined in the scene graph.

**timeFrame:** an array of 32-bit unsigned integer of dimension **numberOfFrames** indicating the absolute rendering time (in milliseconds) for each frame .

NOTE

**numberOfVertices, numberOfNormals, numberOfColors, dimOfOtherAttributes, numberOfOtherAttributes** are instantiated when decoding the static mesh.

### 5.10.6 FAMCAnimationSegmentData class

#### 5.10.6.1 Syntax

```
class FAMCAnimationSegmentData {
    if (animatedFields & 1) {
        FAMCSkinningModel skinningModel;
    }
    FAMCAllResidualErrors allResidualErrors;
}
```

### 5.10.6.2 Semantics

**skinningModel**: contains the skinning model used for motion compensation. This stream is decoded only if vertices coordinates are animated.

**allResidualErrors**: contains the residual errors for all animated attributes (coordinates, normals, colours...).

### 5.10.7 FAMCSkinningModel class

#### 5.10.7.1 Syntax

```
class FAMCSkinningModel {
    FAMCGlobalTranslationDecoder globalTranslationCompensation;
    FAMCAAnimationPartitionDecoder partition;
    FAMCAffineTrasnformsDecoder affineTransforms;
    FAMCAAnimationWeightsDecoder weights;
    if (normalsPredictionStrategy ==3) {
        FAMCVertexInfoDecoder(4, numberOfVertices)normalsPredictors;
    }
}
```

#### 5.10.7.2 Semantics

The **FAMCSkinningModel** class describes the skinning model used for motion compensation. It refers to the following classes:

- **FAMCGlobalTranslationDecoder** class decoding the global translations applied the animated model.
- **FAMCAAnimationPartition** class decoding the segmentation of the mesh vertices into clusters with nearly the some affine motion.
- **FAMCAffineTransforms** class decoding the affine motion of each cluster at each frame.
- **FAMCAAnimationWeights** class decoding the animation weights of the skinning model.
- **FAMCVertexInfoDecoder** class decoding which predictor the decoder should uses for normals. This stream is defined only when normalPred equals 3 (adaptive mode).

### 5.10.8 FAMCGlobalTranslationDecoder

#### 5.10.8.1 Syntax

```
class FAMCGlobalTranslationDecoder {
    FAMCInfoTableDecoder globalTranslationCompensationInfo;
    FamacCabacVx3Decoder2 myGlobalTranslationCompensation(1, numberOfFrames);
}
```

#### 5.10.8.2 Semantics

The **FAMCGlobalTranslationDecoder** class decodes the DCT compressed translations applied to the animated model for motion compensation. In order to recover the original translations values the decoder needs to un-quantize the integer table decoded by the class `globalTranslationCompensation` by using data decoded by the class `FAMCInfoTableDecoder`. An inverse DCT transform should then be applied to the un-quantized real values.

## 5.10.9 FAMCInfoTableDecoder class

### 5.10.9.1 Syntax

```
class FAMCInfoTableDecoder{
    unsigned int(8) numberOfQuantizationBits;
    float(32) maxValueD1;
    float(32) maxValueD2;
    float(32) maxValueD3;
    float(32) minValueD1;
    float(32) minValueD2;
    float(32) minValueD3;
    unsigned char(8) numberOfDecomposedLayers;
    for (int layer = 0; layer < numberOfDecomposedLayers; layer++){
        unsigned int(32) numberOfCoefficientsPerLayer;
    }
}
```

### 5.10.9.2 Semantics

**numberOfQuantizationBits:** a 8-bit unsigned integer indicating the number of quantization bits used.

**maxValueX:** a 32-bit float indicating the maximal value of the Dimension 1 of the encoded three-dimensional real vectors.

**maxValueY:** a 32-bit float indicating the maximal value of the Dimension 2 of the encoded three-dimensional real vectors.

**maxValueZ:** a 32-bit float indicating the maximal value of the Dimension 3 of the encoded three-dimensional real vectors.

**minValueX:** a 32-bit float indicating the minimal value of the Dimension 1 of the encoded three-dimensional real vectors.

**minValueY:** a 32-bit float indicating the minimal value of the Dimension 2 of the encoded three-dimensional real vectors.

**minValueZ:** a 32-bit float indicating the minimal value of the Dimension 3 of the encoded three-dimensional real vectors.

**numberOfDecomposedLayers:** a 8-bit unsigned char indicating the number of sub-tables composing the encoded table.

**numberOfCoefficientsPerLayer:** a 32-bit unsigned integer indicating the number of coefficients for each layer.

The **FAMCInfoTableDecoder** stream describes the information needed to initialize the decoding of a table encoded as **numberOfDecomposedLayers** sub-tables.

## 5.10.10 FAMCCabacVx3Decoder2

### 5.10.10.1 Syntax

```
FAMCCabacVx3Decoder2 ( int V, int F ){
    float(32) delta;
    // read exp-golomb order EGk and unary cut-off
    unsigned int(3) EGk;
    unsigned int(1) cutOff;

    EGk++;
    cutOff++;
}
```

```

// start the arithmetic coding engine
cabac.arideco_start_decoding( cabac._dep );

// decoding of the significance map
CabacContext ccCbp;
CabacContext ccSig[64];
CabacContext ccLast[64];
cabac.biari_init_context( ccCbp, 64 );
for( int i = 0; i < 64; i++ ){
    cabac.biari_init_context( ccSig[i], 64 );
    cabac.biari_init_context( ccLast[i], 64 );
}
bool sigMap[V][F][3];
int cellSize = ( F + 63 ) / 64;
for( int v = 0; v < V; v++ ) {
    for( int c = 0; c < 3; c++ ) {
        if( cabac.biari_decode_symbol( cabac._dep, ccCbp ) ){
            for( int k = 0; k < F; k++ ){
                sigMap[v][k][c] = cabac.biari_decode_symbol( cabac._dep, ccSig[k/cellSize] );
                if( sigMap[v][k][c] && k + 1 < F ){
                    if( cabac.biari_decode_symbol( cabac._dep, ccLast[k/cellSize] ) ){
                        for( int i = k + 1; i < F; i++ ){
                            sigMap[v][i][c] = 0;
                        }
                        break;
                    }
                }
                else if( k + 2 == F ){
                    sigMap[v][k+1][c] = 1;
                }
            }
        }
        else{
            for( int k = 0; k < F; k++ ){
                sigMap[v][k][c] = 0;
            }
        }
    }
}

// decode abs values
CabacContext ccUnary[cutOff];
for( int i = 0; i < cutOff; i++ ) {
    cabac.biari_init_context( ccUnary[i], 64 );
}

int absValues[V][F][3];
for( int v = 0; v < V; v++ ) {
    for( int c = 0; c < 3; c++ ) {
        for( int k = 0; k < F; k++ ) {
            if( sigMap[v][k][c] ){
                int i;
                for( i = 0; i < 16; i++ ){
                    int unaryCtx = ( cutOff - 1 < i ) ? ( cutOff - 1 ) : i;
                    if( 0 == cabac.biari_decode_symbol( cabac._dep, ccUnary[unaryCtx] ) ){
                        break;
                    }
                }
                if( i == 16 ){
                    absValues[v][k][c] += 17 + cabac.exp_golomb_decode_eq_prob( cabac._dep,
EGk );
                }
                else{
                    absValues[v][k][c] = 1 + i;
                }
            }
            else{
                absValues[v][k][c] = 0;
            }
        }
    }
}

```

```

    }
  }
}

// decode signs
int values[V][F][3];
for( int v = 0; v < V; v++ ) {
  for( int c = 0; c < 3; c++ ) {
    for( int k = 0; k < F; k++ ) {
      values[v][k][c] = absValues[v][k][c];
      if( sigMap[v][k][c] ){
        if( cabac.biari_decode_symbol_eq_prob( cabac._dep ) ){
          values[v][k][c] *= -1;
        }
      }
    }
  }
}

// decode predictors
const int PRED_QUANT_BITS = 2;
int pred[V][3];
int predDim[V][3];
int previousDim[3];
pred[0][0] = 0;
pred[0][1] = 0;
pred[0][2] = 0;
previousDim[0] = 1;
previousDim[1] = 1;
previousDim[2] = 1;
CabacContext ccSkip;
CabacContext ccPred;
CabacContext ccPredDim;
cabac.biari_init_context( ccSkip, 64 );
cabac.biari_init_context( ccPred, 64 );
cabac.biari_init_context( ccPredDim, 64 );
for( int v = 1; v < V; v++ ) {
  for( int c = 0; c < 3; c++ ) {
    if( cabac.biari_decode_symbol( cabac._dep, ccSkip ) ){
      pred[v][c] = pred[v-1][c];
      if( pred[v][c] ){
        predDim[v][c] = predDim[v-1][c];
      }
      else{
        predDim[v][c] = 0;
      }
    }
    else{
      pred[v][c] = cabac.unary_exp_golomb_decode( cabac._dep, ccPred, 2 );
      if( pred[v][c] ){
        int predDimRes = cabac.unary_exp_golomb_decode( cabac._dep, ccPredDim, 2 );
        predDimRes <= PRED_QUANT_BITS;
        if( predDimRes ){
          const int largestAllowedPredDim = F + ( 1 << PRED_QUANT_BITS ) - 1;
          if( previousDim[c] + predDimRes > largestAllowedPredDim ) {
            predDimRes *= -1;
          }
          else if( previousDim[c] - predDimRes >= 0 ){
            if( cabac.biari_decode_symbol_eq_prob( cabac._dep ) ){
              predDimRes *= -1;
            }
          }
        }
      }
      predDim[v][c] = predDimRes + previousDim[c];
      previousDim[c] = predDim[v][c];
    }
  }
}
else{

```

```

        predDim[v][c] = 0;
    }
}
}
}
// end the arithmetic coding engine
cabac.biari_decode_final( cabac._dep );
}

```

### 5.10.10.2 Semantics

**delta**: reciprocal value of the quantization step size.

**sigMap[V][F][3]**: array of  $3 * V * F$  bits, indicating the non-zero predicted spectral coefficients of x-, y- and z-component.

**EGk**: order of the Exp-Golomb binarization.

**cutOff**: number of CABAC context models for the unary part of the concatenated unary/ k-th order Exp-Golomb binarization.

**absValues[V][F][3]**: array of  $3 * V * F$  integer values, indicating the absolute values of the predicted spectral coefficients of x-, y- and z-component.

**values[V][F][3]**: array of  $3 * V * F$  integer values, indicating the values of the predicted spectral coefficients including signs of x-, y- and z-component.

**pred[V][3]**: an array indicating the index of the coefficient used for prediction of the current coefficient of x-, y- and z-component.

**predDim[V][3]**: the number of the samples that are used for predicting of x-, y- and z-component.

The FAMCCABACDecoder class decodes a ( $V \times F$ ) array of three dimensional vectors of integer values.

In order to obtain the original values the decoder should inverse the prediction stage as described in the following pseudo-code:

```

// Inverse prediction
for( int v = 1; v < V; v++ ) {
    for( int c = 0; c < 3; c++ ) {
        for( int d = 0; d < predDim[v]; d++ ) {
            if (pred[v] != 0) {
                values[v][d][c] += values[v-pred[v]][d][c];
            }
        }
    }
}
}

```

## 5.10.11 FAMCAnimationPartitionDecoder

### 5.10.11.1 Syntax

```

class FAMCAnimationPartitionDecoder {
    unsigned int(32) numberOfClusters;
    unsigned int(32) compressedPartitionBufferSize;
    FAMCVertexInfoDecoder myFAMCVertexInfoDecoder(numberOfClusters, numberOfVertices);
}

```

### 5.10.11.2 Semantics

**numberOfClusters:** a 32-bit integer indicating the number of motion clusters.

**compressedPartitionBufferSize:** a 32-bit unsigned integer indicating the size of the compressed partition.

The FAMCAAnimationPartition class decodes the segmentation of the mesh vertices into clusters with nearly similar affine motion. It consists of a one dimensional array of integer of length numberOfVertices which assigns to each vertex  $v$  a cluster number partition[ $v$ ]. We refer to Annex I for an example of the encoding process.

### 5.10.12 FAMCVertexInfoDecoder class

#### 5.10.12.1 Syntax

```
class FAMCVertexInfoDecoder (nC, nV){
  // start the arithmetic coding engine
  cabac.arideco_start_decoding( cabac._dep );
  cabac.biari_init_context(cabac._ctx, 61);
  int numberOfBits = (int) (log((double) nC -1)/log(2.0)+ 1.0);
  int occurrence = 0;
  int currentSymbol = 0;
  int v = 0;
  while( v < nV ) {
    currentSymbol = 0;
    for (int pb = numberOfBits - 1; pb >= 0; pb--) {
      int bitOfBitPlane = cabac.biari_decode_symbol_eq_prob(cabac._dep);
      vertexIndex += (bitOfBitPlane * (1<<pb));
    }
    occurrenceMinusOne = cabac.unary_exp_golomb_decode(cabac._dep, cabac._ctx, 2);
    for (int i =0; i < occurrenceMinusOne+1; i++) {
      partition[v] = vertexIndex;
      v++;
    }
  }
  // end the arithmetic coding engine
  cabac.biari_decode_final( cabac._dep );
}
```

#### 5.10.12.2 Semantics

**bitOfBitPlane:** one bit corresponding to the bit of the binary representation of vertexIndex.

**occurrenceMinusOne:** the number minus one of consecutive vertexIndex elements in partition.

FAMCVertexInfoDecoder class decodes, by calling an arithmetic decoder, an array of size numberOfVertices (noted partition). The elements of this array are integers, which are in the range 0, ..., numberOfInfoType -1.

### 5.10.13 FAMCAffineTransformsDecoder class

#### 5.10.13.1 Syntax

```
class FAMCAffineTransformsDecoder{
  FAMCInfoTableDecoder affineTransformsInfo;
  FamcCabacVx3Decoder2 myAffineTransforms(4*numberOfClusters, numberOfFrames);
}
```

### 5.10.13.2 Semantics

The **FAMCAffineTransformsDecoder** class decodes a DCT compressed vertex trajectories. In order to recover the original trajectories the decoder needs to un-quantize the integer table contained in the class `myAffineTransforms` by exploiting the information decoded by the class `AffineTransformsInfo`. An inverse DCT transform should then be applied to the un-quantized real values.

Let  $A_t^k$  be the affine transform associated with the cluster  $k$  at frame  $t$ . In homogeneous coordinates,  $A_t^k$  is given by:

$$A_t^k = \begin{bmatrix} a_t^k & b_t^k & c_t^k & x_t^k \\ d_t^k & e_t^k & f_t^k & y_t^k \\ g_t^k & h_t^k & i_t^k & z_t^k \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the coefficients  $(a_t^k, b_t^k, c_t^k, d_t^k, e_t^k, f_t^k, g_t^k, h_t^k, i_t^k)$  describe the linear part of the affine motion and  $(x_t^k, y_t^k, z_t^k)$  the translational component.

Instead of decompressing the affine transforms assigned to each cluster, the decoder decodes for each cluster  $k$  the trajectories  $M1(k, t)$ ,  $M2(k, t)$ ,  $M3(k, t)$ ,  $M4(k, t)$  of four points defined as follows:

$$M1(k, 0) \in IR^4, M2(k, 0) = M1(k, 0) + \begin{bmatrix} dx \\ 0 \\ 0 \\ 0 \end{bmatrix}, M3(k, 0) = M1(k, 0) + \begin{bmatrix} 0 \\ dy \\ 0 \\ 0 \end{bmatrix}, M4(k, 0) = M1(k, 0) + \begin{bmatrix} 0 \\ 0 \\ dz \\ 0 \end{bmatrix}$$

$$M1(k, t) = A_t^k \times M1(k, 0), M2(k, t) = A_t^k \times M2(k, 0), M3(k, t) = A_t^k \times M3(k, 0), M4(k, t) = A_t^k \times M4(k, 0).$$

In order to compute the sequences of  $(A_t^k)$  the decoder simply apply the following linear equation:

$$A_t^k = [M1(k, 0)M2(k, 0)M3(k, 0)M4(k, 0)]^{-1} \times [M1(k, t)M2(k, t)M3(k, t)M4(k, t)].$$

### 5.10.14 FAMCAAnimationWeightsDecoder class

#### 5.10.14.1 Syntax

```
class FAMCAAnimationWeightsDecoder {
    unsigned int(8) numberOfQuantizationBits;
    float(32) minWeights;
    float(32) maxWeights;
    unsigned int(32) compressedWeightsBufferSize;

    // start the arithmetic coding engine
    cabac.arideco_start_decoding( cabac._dep );
    // decoding retained vertices
    for (int v = 0; v < numberOfVertices; v++) {
        filter[v] = cabac.biari_decode_symbol(cabac._dep, cabac._ctx);
    }

    // decoding clusters adjacency
```

```

for(int k = 0; k < numberOfClusters; k++) {
    int nbrNeighbours = cabac.unary_exp_golomb_decode(cabac._dep, cabac._ctx, 2);
    for(int n = 0; n < nbrNeighbours; n++) {
        for (int bp = numberOfBits-1; bp >= 0; bp--) {
            bool bitOfClusterIndex = cabac.biari_decode_symbol_eq_prob(cabac._dep);
        }
    }
}

// decoding weights
for (int bp = numberOfQuantizationBits -1; bp >= 0; bp--) {
for(int v = 0; v < numberOfVertices; v++) {
    int vertexCluster = partition[v];
    if ( filter[v] == 1) {
        for(int cluster =0; cluster < adj[vertexCluster].size(); cluster++) {
            bool bitOfVertexClusterWeight= cabac.biari_decode_symbol(cabac._dep,
cabac._ctx);
        }
    }
}
}
// end the arithmetic coding engine
cabac.biari_decode_final( cabac._dep );
}

```

#### 5.10.14.2 Semantics

**numberOfQuantizationBits:** a 8-bit unsigned integer indicating the number of quantization bits used for weights.

**compressedWeightsBufferSize:** a 32-bit unsigned integer indicating the compressed stream size.

**minWeights** and **maxWeights:** two 32-bit float indicating the minimal and the maximal values of the animation weights.

**filter:** an array with dimension equals to the number of vertices indicating if a vertex has associated animation weights. If not, the vertex is associated to a single cluster.

The *numberOfBits* is obtained from the **numberOfClusters** as follows:

$$\text{numberOfBits} = (\text{int}) (\log((\text{double}) \text{numberOfClusters}-1)/\log(2.0)+ 1.0);$$

**nbrNeighbours:** an integer indicating the number of neighbors for the current cluster.

**bitOfClusterIndex:** one bit corresponding to the current bitplane of the current cluster index.

**bitOfVertexClusterWeight:** one bit corresponding to the current bitplane of the current weight.

The principle of skinning animation consists in deriving a continuous motion field over the whole mesh, by linearly combining the affine motion of clusters with appropriate weighting coefficients. A skinning model predicts the position  $\hat{\chi}_t^v$  of a vertex  $v$  at frame  $t$  using the following formula:

$$\hat{\chi}_t^v = \sum_{k=1}^{\text{numberOfClusters}} \omega_k^v A_t^k \chi_1^v$$

where  $\omega_k^v$  is a coefficient that controls the motion influence of the cluster  $k$  over the vertex  $v$ .  $A_t^k$  represents the affine transform associated with the cluster  $k$  at frame  $t$  expressed in homogeneous coordinates.

The optimal weight vector  $\omega^v = (\omega_k^v)_{k \in \{1, \dots, \text{numberOfClusters}\}}$  is computed at the encoder side and sent to the decoder. The  $\omega_k^v = 0$  when  $k$  is not a neighbour of the cluster that  $v$  belongs.

The decoding process is composed of three steps:

- (a) vertices selection decoding,
- (b) clusters adjacency decoding, and
- (c) weights decoding.

#### a) Vertices selection decoding

First, the CABAC context is initialized with value 61. Then, the one dimensional array **filter** of size **numberOfVertices** is decoded by using the CABAC function `biari_decode_symbol`.

#### b) Clusters adjacency decoding

The CABAC context is initialized with value 61. For each cluster  $k$ , the number of its neighbours is decoded by using the CABAC function `unary_exp_golomb_decode`. Then, the index of each neighbour is decoded by using the CABAC function `biari_decode_symbol_eq_prob`. Each index is represented by its binary representation on **numberOfBits** bits.

#### c) Weights decoding

The CABAC context is initialized with value 2. The weights are decoded bit-plane per bit-plane using the CABAC function `biari_decode_symbol`. In order to retrieve the values of weights, the quantization process needs to be reversed.

### 5.10.15 FAMCAIIResidualErrors

#### 5.10.15.1 Syntax

```
class FAMCAIIResidualErrors {
    switch (transformType) {
        case 0 : // Lifting
        case 1 : // DCT
            if (animatedFields & 1)
                FAMCAIIResidualErrorsInfo coordResidualErrorsInfo;
            if (animatedFields & 2)
                FAMCAIIResidualErrorsInfo normalResidualErrorsInfo;
            if (animatedFields & 4)
                FAMCAIIResidualErrorsInfo colorResidualErrorsInfo;
            if (animatedFields & 8)
                FAMCAIIResidualErrorsInfo otherAttributesResidualErrorsInfo;
        do {
            if (animatedFields & 1)
                FAMCCabacVx3Decoder2 coordErrorsLayerLift(numberOfVertices,
                    numberOfCoefficientsPerLayer);
            if (animatedFields & 2)
                FAMCCabacVx3Decoder2 normalErrorsLayerLift(numberOfNormals,
                    numberOfCoefficientsPerLayer);
            if (animatedFields & 4)
                FAMCCabacVx3Decoder2 colorErrorsLayerLift(numberOfColors,
                    numberOfCoefficientsPerLayer);
            if (animatedFields & 8)
                FAMCCabacVx3Decoder2
                    otherAttributesErrorsLayerLift(numberOfOtherAttributes,
                    numberOfCoefficientsPerLayer);
            bit(32)* next;
        }
        while (next==FAMCAIIResidualErrorsInfoStartCode);
        break;
    case 2: // LD
```

```

        FAMCLDDecoder allErrorsLD(numberOfVertices, numberOfFrames, animatedFields);
        break;
    case 3: // Lift + LD
        FAMCLDDecoder allErrorsLiftLD(numberOfVertices, numberOfFrames, animatedFields);
        break;
    case 4: // DCT + LD
        FAMCLDDecoder allErrorsDCTLD(numberOfVertices, numberOfFrames, animatedFields);
        break;
}
}

```

### 5.10.15.2 Semantics

**coordResidualErrorsInfo**: a FAMCInfoTableDecoder class decoding the information needed to initialize the decoding process for coordinates residual errors.

**normalsResidualErrorsInfo**: a FAMCInfoTableDecoder class decoding the information needed to initialize the decoding process for normals residual errors.

**colorResidualErrorsInfo**: a FAMCInfoTableDecoder class decoding the information needed to initialize the decoding process for colours residual errors.

**otherAttributesResidualErrorsInfo**: a FAMCInfoTableDecoder class decoding the information needed to initialize the decoding process for other attributes residual errors.

**coordErrorsLayer** : a FAMCCabacVx3Decoder2 class decoding a sub-table of integer of dimension numberOfVertices x numberOfCoefficientsPerLayer.

**normalErrorsLayer** : a FAMCCabacVx3Decoder2 class decoding a sub-table of integer of dimension numberOfNormals x numberOfCoefficientsPerLayer.

**colorErrorsLayer** : a FAMCCabacVx3Decoder2 class decoding a sub-table of integer of dimension numberOfColors x numberOfCoefficientsPerLayer.

**otherAttributesErrorsLayer** : a FAMCCabacVx3Decoder2 class decoding a sub-table of integer of dimension numberOfOtherAttributes x numberOfCoefficientsPerLayer.

**allErrorsLD** : a FAMCLDDecoder class decoding LD prediction errors (as an array of integers of dimension 3 x numberOfVertices x numberOfFrames) and auxiliary data, which are needed for reconstruction of coordinates, normals, colors, and other attributes.

**allErrorsLiftLD** : a FAMCLDDecoder class decoding LD prediction errors of lifting coefficients (an array of integers of dimension 3 x numberOfVertices x numberOfFrames) together with auxiliary data, which are needed for reconstruction of lifting coefficients corresponding to coordinates, normals, colors, and other attributes. With a subsequent inverse lifting transform coordinates, normals, colors, and other attributes are obtained.

**allErrorsDCTLD** : a FAMCLDDecoder class decoding LD prediction errors of DCT coefficients (an array of integers of dimension 3 x numberOfVertices x numberOfFrames) together with auxiliary data, which are needed for reconstruction of DCT coefficients corresponding to coordinates, normals, colors, and other attributes. With a subsequent inverse DCT coordinates, normals, colors, and other attributes are obtained.

Each decoded layer with transformType 0 or 1 contains a subset of the spectrum coefficients, arranged from low frequency (layer 0) to high frequency (layer n). After decoding a layer, the tables for each component (coordinates, normals, colors, other attributes) are concatenated. The values of a layer superior to the current one are assumed to be zero.

To recover the original values the decoder applies:

- An inverse Lift transform followed by an un-quantization when transformType is Lift,
- An un-quantization followed by an inverse DCT when transformType is DCT.

The coordinate residual errors are decompressed and stored as set of vectors

$$\left( \varepsilon_t^v \right)_{\substack{v \in \{1, \dots, \text{numberOfCoord}\} \\ t \in \{2, \dots, \text{numberOfCoordKeys}\}}}$$

expressed in homogeneous coordinates. The decoder computes the position  $\chi_t^v$  of a vertex  $\mathbf{v}$  at frame  $\mathbf{t}$  by applying the following formula:

$$\chi_t^v = \sum_{k=1}^{\text{numberOfClusters}} \omega_k^v A_t^k \chi_1^v + \gamma_t + \varepsilon_t^v,$$

where  $\chi_1^v$  is the position of vertex  $\mathbf{v}$  at the first frame,  $A_t^k$  is the affine transform associated with the cluster  $\mathbf{k}$  at frame  $\mathbf{t}$  expressed in homogeneous coordinates,  $\omega_k^v$ : the weight of cluster  $\mathbf{k}$  at vertex  $\mathbf{v}$ ,  $\gamma_t$ : the global motion of frame  $\mathbf{t}$ ,  $\varepsilon_t^v$ : coordinate residual errors of vertex  $\mathbf{v}$  at frame  $\mathbf{t}$ .

The normal residual errors are decompressed and stored as set of vectors

$$\left( n_t^v \right)_{\substack{v \in \{1, \dots, \text{numberOfCoord}\} \\ t \in \{2, \dots, \text{numberOfCoordKeys}\}}}$$

expressed in homogeneous coordinates. The decoder computes the normal  $N_t^v$  of a vertex  $\mathbf{v}$  at frame  $\mathbf{t}$  by applying on of the following equations

$$N_t^v = N_1^v + n_t^v \text{ if normalsPredictionStrategy}=0$$

$$N_t^v = \sum_{k=1}^{\text{numberOfClusters}} \omega_k^v A_t^k N_1^v + n_t^v \text{ if normalsPredictionStrategy}=1$$

$$N_t^v = \frac{U_t^v \times W_t^v}{\|U_t^v \times W_t^v\|} + n_t^v \text{ if normalsPredictionStrategy}=2$$

where

$$U_t^v = \frac{\sum_{k=1}^{\text{numberOfClusters}} \omega_v^k A_t^k U^v}{\left\| \sum_{k=1}^K \omega_v^k A_t^k U^v \right\|}, \quad W_t^v = \frac{\sum_{k=1}^{\text{numberOfClusters}} \omega_v^k A_t^k W^v}{\left\| \sum_{k=1}^K \omega_v^k A_t^k W^v \right\|}, \quad (U^v, W^v, N_1^v) \text{ is the orthonormal basis of } IR^3,$$

$N_1^v$  is the normal of vertex  $\mathbf{v}$  at the first frame and  $n_t^v$  is the normal residual errors of vertex  $\mathbf{v}$  at frame  $\mathbf{t}$ .

The colour residual errors are decompressed and stored as set of vectors

$$\left( c_t^v \right)_{\substack{v \in \{1, \dots, \text{numberOfCoord}\} \\ t \in \{2, \dots, \text{numberOfCoordKeys}\}}}.$$

The decoder computes the colour  $C_t^v$  of a vertex  $v$  at frame  $t$  by applying equation

$$C_t^v = C_1^v + c_t^v$$

where  $C_1^v$  is the colour of vertex  $v$  at the first frame and  $c_t^v = \begin{pmatrix} R_t^v \\ G_t^v \\ B_t^v \end{pmatrix}$  is colour residual errors of vertex  $v$  at frame  $t$ .

The other attributes decoding is identical to normal decoding.

## 5.10.16 FAMCLDDecoder class

### 5.10.16.1 Syntax

```
class FAMCLDDecoder (nV, nF, fields){
    bit(1) newLayeredDecompositionNeeded;
    bit(1) layeredDecompositionIsEncoded;
    bit(6) bitsNotDefined;

    if (newLayeredDecompositionNeeded){
        unsigned int(32) numberOfDecomposedLayers;
        if (layeredDecompositionIsEncoded){
            FAMCLayeredDecompositionDecoder myFAMCLayeredDecompositionDecoder
(numberOfDecomposedLayers, nV);
        }
    }

    unsigned int(32) numberOfEncodedLayers;
    if (animatedFields & 1) float(64) coordsQuantizationStepLD;
    if (animatedFields & 2) float(64) normalsQuantizationStepLD;
    if (animatedFields & 4) float(64) colorsQuantizationStepLD;
    if (animatedFields & 8) float(64) otherAttributesQuantizationStepLD;

    for (int frameNumberDec=0; frameNumberDec<nF; ++frameNumberDec){
        FAMCLDFrameHeaderDecoder myFAMCLDFrameHeaderDecoder;

        hasCoordsPredBits =
            ((coordsPredictionModeLD == 3) || (coordsPredictionModeLD == 4)) ? 1 : 0;

        hasNormalsPredBits =
            ((normalsPredictionModeLD == 3) || (normalsPredictionModeLD == 4)) ? 1 : 0;

        hasColorsPredBits =
            ((colorsPredictionModeLD == 3) || (colorsPredictionModeLD == 4)) ? 1 : 0;

        hasOtherAttributesPredBits =
            ((otherAttributesPredictionModeLD == 3) || (otherAttributesPredictionModeLD ==
4)) ? 1 : 0;

        unsigned int(32) compressedFrameSizeLD;
        for (int layerNumber=0; layerNumber<numberOfEncodedLayers; ++layerNumber){
            if (animatedFields & 1)
                FAMCCabacVx3Decoder
                    resCoords(numberOfVerticesInLayer[layerNumber], hasCoordsPredBits);
            if (animatedFields & 2)
                FAMCCabacVx3Decoder
                    resNormals(numberOfVerticesInLayer[layerNumber], hasNormalsPredBits);
            if (animatedFields & 4)
```

```

    FAMCCabacVx3Decoder
        resColors(numberOfVerticesInLayer[layerNumber], hasColorsPredBits);
    if (animatedFields & 8)
        FAMCCabacVx3Decoder resOtherAttributes(numberOfVerticesInLayer[layerNumber],
        hasOtherAttributesPredBits);
    }
}
}

```

### 5.10.16.2 Semantics

**newLayeredDecompositionNeeded:** one bit indicating if in the current segment a new layered decomposition is needed. In such case (newLayeredDecompositionNeeded equals 1) the decoded decomposition becomes the current decomposition, which is used in the current and following segments.

**layeredDecompositionIsEncoded:** one bit indicating if a layered decomposition is encoded in the bit-stream. If not, the layered decomposition is determined using the deterministic algorithm presented in Annex K.

**bitsNotDefined:** 6-bits with not defined semantics. Reserved for ISO purposes.

**numberOfDecomposedLayers:** a 32-bit unsigned integer indicating the number of layers created during layered decomposition. This may be different from the number of layers that are present in the bitstream.

**numberOfEncodedLayers:** a 32-bit unsigned integer smaller or equal than numberOfDecomposedLayers indicating the number of layers encoded in the stream.

**coordsQuantizationStepLD:** a 64-bit float specifying the quantization step for coordinates.

**normalsQuantizationStepLD:** a 64-bit float specifying the quantization step for normals.

**colorsQuantizationStepLD:** a 64-bit float specifying the quantization step for colors.

**otherAttributesQuantizationStepLD:** a 64-bit float specifying the quantization step for other attributes.

**compressedFrameSizeLD:** a 32-bit unsigned integer indicating the size of the compressed frame.

**resCoords:** a FAMCCabacVx3Decoder class decoding a table of integers of dimension numberOfVerticesInLayer[layerNumber] x 3 corresponding to quantized prediction errors of coordinates. If hasCoordsPredBits equals 1 additionally an array of bits of size numberOfVerticesInLayer[layerNumber] is decoded.

**resNormals:** a FAMCCabacVx3Decoder class decoding a table of integers of dimension numberOfVerticesInLayer[layerNumber] x 3 corresponding to quantized prediction errors of normals. If hasNormalsPredBits equals 1 additionally an array of bits of size numberOfVerticesInLayer[layerNumber] is decoded.

**resColors:** a FAMCCabacVx3Decoder class decoding a table of integers of dimension numberOfVerticesInLayer[layerNumber] x 3 corresponding to quantized prediction errors of colors. If

hasColorsPredBits equals 1 additionally an array of bits of size numberOfVerticesInLayer[layerNumber] is decoded.

**resOtherAttributes:** a FAMCCabacVx3Decoder class decoding a table of integers of dimension numberOfVerticesInLayer[layerNumber] x 3 corresponding to quantized prediction errors of other attributes. If hasOtherAttributesPredBits equals 1 additionally an array of bits of size numberOfVerticesInLayer[layerNumber] is decoded.

The **FAMCLDDecoder** class describes vertex coordinates, normals, colors, and other attributes of an animation segment. The process of obtaining this data is described below.

The **FAMCLDDecoder** class decodes a new layered decomposition maximally once per animation segment (if newLayeredDecompositionNeeded equals 1). Thereby, either data is decoded (layeredDecompositionIsEncoded equals 1) and used together with the mesh connectivity to describe a layered decomposition, or a new layered decomposition is determined based only on connectivity (layeredDecompositionIsEncoded equals 0). The process of deriving a layered decomposition is described in detail in Annex K.

The layered decomposition Id[][] is used for guiding the process of predictive reconstruction of 3D coordinates using decoded quantized prediction errors.

For each frame (frameNumberDec) a frame header is decoded. Thereby the following values are decoded or computed:

- frameType,
- frameNumberDis, refFrameNumberOffsetDis0, and refFrameNumberOffsetDis1,
- coordsPredictionModeLD, normalsPredictionModeLD, colorsPredictionModeLD, otherAttributesPredictionModeLD.

Subsequently, quantized 3D prediction errors of coordinates, noted resCoord[frameNumberDis][layerNumber][c] with c=0, ..., numberOfVerticesInLayer[layerNumber] are decoded for each layer in a frame. Here, the value numberOfVerticesInLayer[layerNumber] is obtained from the current layered decomposition.

Furthermore, depending on the prediction mode value for coordinates (coordsPredictionModeLD), which is decoded frame-wisely, each resCoords[frameNumberDis][layerNumber][c] gets a distinct prediction type noted coordsPredType[frameNumberDis][layerNumber][c] as specified in Table AMD2.7.

**Table AMD2.7 — The correspondences between predictionModeLD values and prediction types predType.**

coordsPredictionModeLD value\ normalsPredictionModeLD value\ colorsPredictionModeLD value\ otherAttributesPredictionModeLD value	coordsPredType value\ normalsPredType value\ colorsPredType value\ otherAttributesPredType value	Predictor name
0	0	Delta
1	1	Linear
2*	2	Non-linear
3*	1 or 2	Linear or non-linear, adaptiv
4	0 or 1	Delta or linear, adaptiv
5-15	Not defined	-

\*NOTE For normals, colors and other attributes prediction modes 2 and 3 are not allowed.

If coordPredictionModeLD=0,1,2 then

$\text{coordsPredType}[\text{frameNumberDis}][\text{layerNumber}][c] = \text{coordPredictionModeLD}$

for all c and layers of a frame with frame number frameNumberDis.

If coordPredictionModeLD =3 ,4, two values are possible for coordsPredType. In this case value hasCoordsPredBits is equal to 1 and the **FAMCCabacVx3Decoder** decodes additionally to quantized residuals also an array of bits  $\text{coordsPredBits}[\text{frameNumberDis}][\text{layerNumber}][c]$ , which is used to derive definite prediction types.

If coordsPredictionModeLD=3 then

$\text{coordsPredType}[\text{frameNumberDis}][\text{layerNumber}][c] = 1$  if  $\text{coordsPredBits}[\text{frameNumberDis}][\text{layerNumber}][c] == 1$  and

$\text{coordsPredType}[\text{frameNumberDis}][\text{layerNumber}][c] = 2$  if  $\text{coordsPredBits}[\text{frameNumberDis}][\text{layerNumber}][c] == 0$ .

If coordsPredictionModeLD=4 then

$\text{coordsPredType}[\text{frameNumberDis}][\text{layerNumber}][c] = \text{coordPredBits}[\text{frameNumberDis}][\text{layerNumber}][c]$ .

After decoding quantized prediction errors of coordinates of a layer in a frame ( $\text{resCoords}[\ ][\ ]$ ) and assigning prediction types ( $\text{coordsPredType}[\ ][\ ]$ ), quantized prediction errors of normals ( $\text{resNormals}[\ ][\ ]$ ), colors ( $\text{resColors}[\ ][\ ]$ ), and other attributes ( $\text{resOtherAttributes}[\ ][\ ]$ ) are decoded. Similar to  $\text{resCoords}[\ ][\ ]$  also  $\text{resNormals}[\ ][\ ]$ ,  $\text{resColors}[\ ][\ ]$ , and  $\text{resOtherAttributes}[\ ][\ ]$  get prediction types  $\text{normalsPredType}[\ ][\ ]$ ,  $\text{colorsPredType}[\ ][\ ]$ , and  $\text{otherAttributesPredType}[\ ][\ ]$  assigned Table AMD2.7).

Finally, for each quantized prediction error `resCoords[frameNumberDis][layerNumber][c]`, `resNormals[frameNumberDis][layerNumber][c]`, `resColors[frameNumberDis][layerNumber][c]`, and `resOtherAttributes[frameNumberDis][layerNumber][c]`, a corresponding value is reconstructed as follows:

```
for (int frameNumberDec=0; frameNumberDec<numberOfFrames; ++frameNumberDec){
    frameNumberDis = frameNumberDec2DisList[frameNumberDec];
    for (int layer=0; layer<numberOfEncodedLayer; ++layer){
        for (int c=0; c<numberOfVerticesInLayer[layer]; c++){
            // reconstruct coordinate corresponding to vertex ld[layer][c].to
        }
        for (int c=0; c<numberOfVerticesInLayer[layer]; c++){
            // reconstruct normal corresponding to vertex ld[layer][c].to
        }
        for (int c=0; c<numberOfVerticesInLayer[layer]; c++){
            // reconstruct color corresponding to vertex ld[layer][c].to
        }
        for (int c=0; c<numberOfVerticesInLayer[layer]; c++){
            // reconstruct other attributes corresponding to vertex ld[layer][c].to
        }
    }
}
```

Here, the `frameNumberDec2DisList` is obtained from data decoded by the `FAMCLDFrameHeaderDecoder` class. See Annex L for a detailed description of the reconstruction process of coordinates, normals, colors, and other attributes.

Finally, all coordinates, normals, colors and other attributes of an animation segment are decoded.

### 5.10.17 FAMCLayeredDecompositionDecoder class

#### 5.10.17.1 Syntax

```
class FAMCLayeredDecompositionDecoder (L,V){
    unsigned int(32) compressedPartitionBufferSize;
    FAMCVertexInfoDecoder MyFAMCVertexInfoDecoder(L, V);
    unsigned int(32) compressedSimplificationBufferSize;
    for (int layer=L-1; layer>=1; --layer){
        FAMCSimplificationModeDecoder myFAMCSimplificationModeDecoder
(numberOfVerticesInLayer[layer]);
    }
}
```

#### 5.10.17.2 Semantics

**compressedPartitionBufferSize:** a 32 bit unsigned integer indicating the compressed stream size of the partition in layers.

**compressedSimplificationBufferSize:** a 32 bit unsigned integer indicating the compressed stream size.

The `FAMCLayeredDecompositionDecoder` class decodes a sequence of simplification operations. Each simplification operation is represented as a couple (*vertexIndex, mode*), both values are unsigned integers.

First, the `FAMCVertexInfoDecoder` class decodes an array (noted partition) of integers of size V. Each element `partition[v]` of the array is in the range 0,...L-1 and indicates the assignment of vertex v to layer `partition[v]`.

The array of integers `numberOfVerticesInLayer` is obtained from the decoded array partition. The derivation process is illustrated with the following pseudo code:

```

int numberOfVerticesInLayer[L];
for (v=0; v<V; ++v){
    numberOfVerticesInLayer[partition[v]]++;
}

```

The **FAMCSimplificationModeDecoder** decodes an array `simplificationMode[layer][c]` of simplification modes for `layer=L-1,...,1` and `c=0,..., numberOfVerticesInLayer[layer]-1`. The following pseudo code illustrates how an array of simplification operations (noted `vssop`) is obtained from the arrays `partition` and `simplificationMode`:

```

struct SimplificationOperation{
    int vertexIndex;
    int mode;
};

vector< vector<SimplificationOperation> > vvsop(L);
for (int v=0; v<V; ++v){
    SimplificationOperation sop;
    sop.vertexIndex = v;
    vvsop[partition[v]].push_back(sop);
}
for (int layer=L-1; layer>=1; --layer){
    for (int c=0; c<numberOfVerticesInLayer[layer]; ++c){
        vvsop[layer][c].mode = simplificationMode[layer][c];
    }
}

```

By reorganizing the data contained in `partition` and `simplificationMode`, the simplification operations

```
(vvsop[layer][c].vertexIndex, vvsop[layer][c].mode)
```

for

```

layer=1,...,L-1 and
c=0,..., numberOfVerticesInLayer[layer].

```

are obtained. The procedure for building the layer decomposition from simplification operations is described in Annex K.

### 5.10.18 FAMCSimplificationModeDecoder class

#### 5.10.18.1 Syntax

```

class FAMCSimplificationModeDecoder(V){
    // start the arithmetic coding engine
    cabac.arideco_start_decoding( cabac._dep );
    cabac.biari_init_context(cabac._ctx, 61);
    for (int c=0; c<V; ++c){
        simplificationMode[c] = cabac.unary_exp_golomb_decode(cabac._dep, cabac._ctx, 2);
    }
    // end the arithmetic coding engine
    cabac.biari_decode_final( cabac._dep );
}

```

#### 5.10.18.2 Semantics

**simplificationMode**: an arithmetic encoded integer indicating a simplification mode.

The **FAMCSimplificationModeDecoder** class decodes an array of size V of unsigned integers.

### 5.10.19 FAMCLDFrameHeaderDecoder class

#### 5.10.19.1 Syntax

```
class FAMCLDFrameHeaderDecoder {
    signed   int(8)  frameNumberOffsetDis;
    unsigned int(2)  frameType; //I, P, or B-frame
    unsigned int(7)  refFrameNumberOffsetDec0;
    unsigned int(7)  refFrameNumberOffsetDec1;
    unsigned int(4)  coordsPredictionModeLD;
    unsigned int(4)  normalsPredictionModeLD;
    unsigned int(4)  colorsPredictionModeLD;
    unsigned int(4)  otherAttributesPredictionModeLD;
}
```

#### 5.10.19.2 Semantics

**frameNumberOffsetDis:** an 8-bit integer used to compute the current frame number in display order (noted **frameNumberDis**).

$\text{frameNumberDis} = \text{frameNumberPrevDis} + \text{frameNumberOffsetDis}$ ,

where **FrameNumberPrevDis** is the frame number in display order of the last decoded frame. For the first decoded frame of a segment **frameNumberPrevDis** equals 0.

**frameType:** a 2-bit integer indicating the frame type of the currently decoded frame: 0=I-frame, 1=P-frame, 2=B-frame.

**refFrameNumberOffsetDec0:** a 7-bit integer used to compute a reference frame number in decoding order (noted **refFrameNumberDec0**). **refFrameNumberDec0** is computed only for P and B frames as follows:

$\text{refFrameNumberDec0} = \text{frameNumberDec} - \text{refFrameNumberOffsetDec0}$ .

**refFrameNumberOffsetDec1:** a 7-bit integer used to compute a second reference frame number in decoding order (noted **refFrameNumberDec1**). **refFrameNumberDec1** is computed only for B frames as follows:

$\text{refFrameNumberDec1} = \text{frameNumberDec} - \text{refFrameNumberOffsetDec1}$ .

**coordsPredictionModeLD:** a 4-bit integer specifying the prediction mode used for predicting coordinates.

**normalsPredictionModeLD:** a 4-bit integer specifying the prediction mode used for predicting normals.

**colorsPredictionModeLD:** a 4-bit integer specifying the prediction mode used for predicting colors.

**otherAttributesPredictionModeLD:** a 4-bit integer specifying the prediction mode used for predicting otherAttributes.

During decoding the list **frameNumberDec2DisList** is updated as follows:

$\text{frameNumberDec2DisList}[\text{frameNumberDec}] = \text{frameNumberDis}$

Note: due to the usage of 7-bit integers for **refFrameNumberOffsetDec0** and **refFrameNumberOffsetDec1** only frames which are within the last 128 decoded frames can be used as reference frames.

## 5.10.20 FAMCCabacVx3Decoder

### 5.10.20.1 Syntax

```

class FAMCCabacVx3Decoder (V, B){
    // decoding the significance map
    unsigned int(8) sigMapInitProb;
    // start the arithmetic coding engine
    cabac.arideco_start_decoding( cabac._dep );
    cabac.biari_init_context(cabac._ctx, sigMapInitProb);
    for (int v=0; v < V; ++v){
        for (int d = 0; d < 3; d++){
            sigMap[v][d] = cabac.biari_decode_symbol(cabac._dep, cabac._ctx);
        }
    }

    //decoding signs
    for (int v = 0; v < V; v++){
        for (int d = 0; d < 3; d++){
            if (sigMap[v][d] == 1) {
                negative[v][d] = cabac.biari_decode_symbol_eq_prob(cabac._dep);
            }
        }
    }

    // end the arithmetic coding engine
    cabac.biari_decode_final( cabac._dep );
    //decoding abs values
    unsigned int(8) absInitProb;
    unsigned int(8) numberOfAbsValuesBitPlanes;

    // start the arithmetic coding engine
    cabac.arideco_start_decoding( cabac._dep );

    cabac.biari_init_context(cabac._ctx, absInitProb);
    for (int pb = numberOfAbsValuesBitPlanes - 1; pb >= 0; pb--) {
        for (int v = 0; v < V; v++){
            for (int d = 0; d < 3; d++){
                if (sigMap[v][d] == 1)
                    if (cabac.biari_decode_symbol(cabac._dep, cabac._ctx)){
                        values[v][d] += (1<<pb);
                    }
                if (pb == 0) {
                    values[v][d]++;
                    values[v][d] = (negative[v][d]) ? -values[v][d] : values[v][d];
                }
            }
        }
    }

    //end the arithmetic coding engine
    cabac.biari_decode_final( cabac._dep );

    if (B==1){
        //decoding prediction modes
        unsigned int(8) predModeInitProb;
        // start the arithmetic coding engine
        cabac.arideco_start_decoding( cabac._dep );
        cabac.biari_init_context(cabac._ctx, predModeInitProb);
        for (int v=0; v < V ++v){
            bits[v] = cabac.biari_decode_symbol(cabac._dep, cabac._ctx);
        }
        // end the arithmetic coding engine
        cabac.biari_decode_final( cabac._dep );
    }
}

```

5.10.20.2 Semantics

**sigMapInitProb:** a 8-bit integer indicating the initial value for a CABAC context for significance map decoding.

**signMap:** an arithmetic encoded array of size  $V \times 3$  of bits indicating the non-zero values.

**negative:** an arithmetic encoded array of size  $V \times 3$  of bits indicating negative and positive values (negative=1).

**absInitProb:** a 8-bit integer indicating the initial value for a CABAC context for absolute values decoding.

**numberOfAbsValuesBitPlanes:** an 8-bit integer indicating the number of bit-planes to decode for decoding of absolute values.

**values:** an arithmetic encoded array of size  $V \times 3$  of integers values.

**predModelInitProb:** a 8-bit integer indicating the initial value for a CABAC context for decoding of bits (decoded only if parameter B equals 1).

**bits:** an array of size V of bits (decoded only if parameter B equals 1).

The **FAMCCabacVx3Decoder** class decoded an array of size  $V \times 3$  of integers. If B equals 1 an array of size V of bits is decoded as well.

In Annex G, replace the table with the following:

Company	Address
Fraunhofer Institute for Telecommunications, Heinrich-Hertz-Institut	Einsteinufer 37, 10587 Berlin, Germany
Fraunhofer-Gesellschaft	Postfach 20 07 33, 80007 Munich, Germany
IMEC	Kapeldreef 75, 3001 Leuven, Belgium
Leibniz Universität Hannover, Institut für Informationsverarbeitung (TNT)	Appelstr. 9A, Hannover 30167 Germany
Samsung Electronics Co. Ltd.	
Superscape	

After Annex H, add the following new annexes:

## Annex I (informative)

### Partition Encoding

The **partition** table is encoded by using a Run Length Encoding (RLE) strategy followed by the CABAC encoding. The table is encoded as set of pairs (*symbol, occurrence*) as illustrated Figure I.1. Here, the pair (*symbol, occurrence*) is binarized (fixed-length binary representation for the symbol and a concatenated unary/exponential Golomb code of order 0 for the occurrence) and encoded by using CABAC.

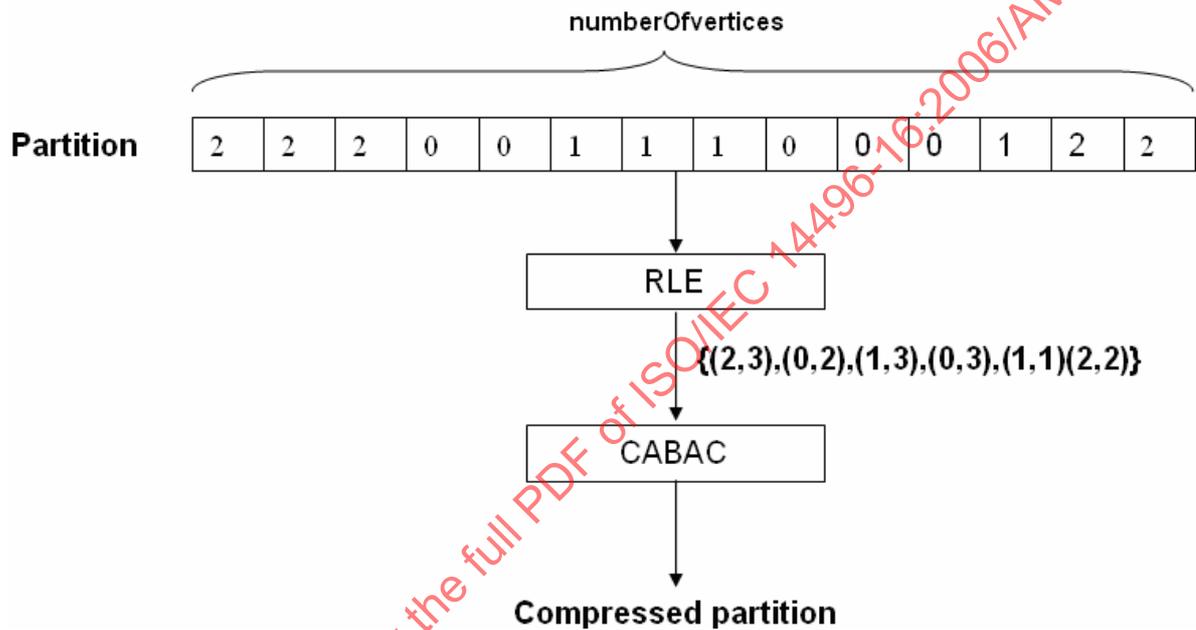


Figure I.1— RLE encoding of the vertices partition: example with 14 vertices and 3 clusters.

## Annex J (informative)

### Animation Weights Encoding

In order to encode the animation weights, the encoder proceeds as follows (Figure J.1):

- A subset of vertices with multiple influences is selected and compressed using CABAC. The selection procedure consists in comparing, for a given vertex  $v$  belonging to the cluster  $k$ , the performances of the skinning model with optimal weights with those obtained by using the unitary weights  $\omega_k^v = \delta_{k,l}, \forall l \in \{1, \dots, \text{numberOfClusters}\}$  ( $\delta_{k,l}$  being the Kronecker symbol).

**Remark: The unitary weights may be derived directly from the vertices partition and should not be sent to the decoder.**

- The clusters adjacency  $(N(k))_{k \in \{1, \dots, \text{numberOfClusters}\}}$  is compressed by using CABAC. More precisely, for each cluster  $k$ , the number of its neighbors and their indices are encoded. Here, the number of neighbors is binarized by using concatenated unary/exponential Golomb codes of order 0 and the indices are expressed in a fixed-length binary representation.
- The weights of the retained vertices are uniformly quantized on **numberOfQuantizationBits**, binarized into bit planes and compressed by using CABAC.

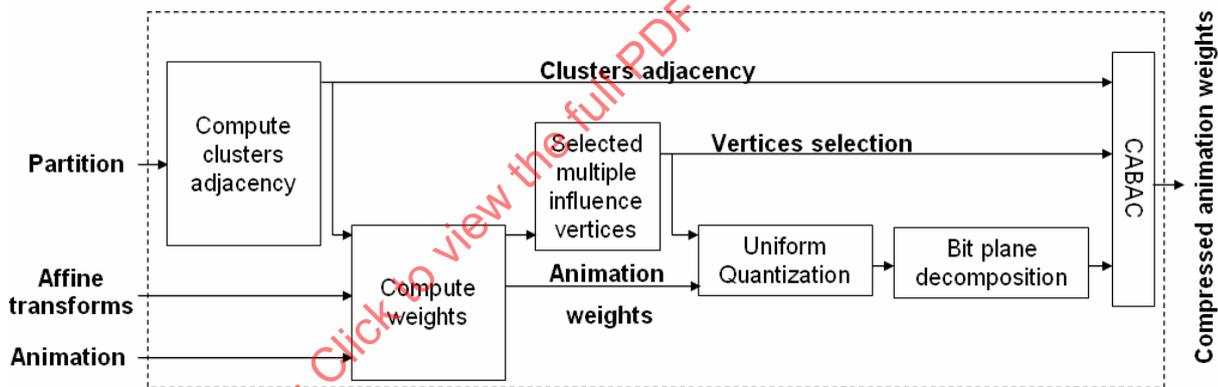


Figure J.1 — Animation weights encoding.

## Annex K (normative)

### Layered decomposition

#### K.1 Obtaining the layered decomposition

A layered decomposition  $ld[][]$  consists of `numberOfDecomposedLayers` arrays

$ld[l]$

with  $l=0, \dots, \text{numberOfDecomposedLayers}-1$  of (in general) different sizes. Each array  $ld[l]$  is of size `numberOfVerticesInLayer[l]` and stores elements of type `VertexContext`:

```
class VertexContext{
    int to;
    int from[];
}
```

The following function returns true, if a layered decomposition  $ld[][]$  has valid sizes:

```
bool areSizesValid(ld[][]){
    if (ld.size()!=numberOfDecomposedLayers) return false;
    for (int l=0; l<numberOfDecomposedLayers; ++l){
        if (ld[l].size()!=numberOfVerticesInLayer[l]) return false;
    }
    return true;
}
```

Each vertex context element of a layered decomposition  $ld[l][c]$  has a member variable  $ld[l][c].to$ , which stores a vertex index, and a member variable  $ld[l][c].from$ , which stores an array of vertex indices. Vertex indices in  $ld[l][c].from$  signify vertices which are used for prediction of a 3D coordinate, which is assigned to vertex  $ld[l][c].to$ . Hence, 3D coordinates assigned to vertex indices in  $ld[l][c].from$  have to be decoded before decoding  $ld[l][c].to$ . The following function returns true if  $ld[][]$  satisfies this property:

```
bool isContentValid(ld[][]){
    vector<bool> isDecoded(numberOfVertices, false);
    for (l=0; l<numberOfDecomposedLayers; ++l){
        for (int c=0; c<numberOfVerticesInLayer[l]; ++c){
            for (int k=0; k<ld[l][c].from.size(); ++k){
                if (!isDecoded[ld[l][c].from[k]]) return false;
            }
            isDecoded[ld[l][c].to]=true;
        }
    }
    return true;
}
```

In the process of obtaining a layered decomposition, the layers are derived in the following order:

$ld[\text{numberOfDecomposedLayers}-1], \dots, ld[0]$ .

There exist two ways to derive layers  $ld[\text{numberOfDecomposedLayers}-1], \dots, ld[1]$ :

- using decoded data and mesh connectivity (`layeredDecompositionIsEncoded` equals 1),
- using only mesh connectivity (`layeredDecompositionIsEncoded` equals 0)

Layer  $ld[0]$  is always derived by using only mesh connectivity.

Mesh connectivity is described by three arrays:

- an array  $fv[][]$  of integers of size  $numberOfFaces \times 3$ ,
- an array  $isVertexDeleted[]$  of boolean values of size  $numberOfVertices$ .
- an array  $isFaceDeleted[]$  of boolean values of size  $numberOfFaces$ .

The last two arrays specify if a vertex or a face, respectively, is already deleted. Initially, all elements of this arrays are set to false. Each layer  $ld[l]$  is derived by processing and manipulating arrays  $fv[][]$ ,  $isVertexDeleted[]$ , and  $isFaceDeleted[]$  (if  $layeredDecompositionIsEncoded$  equals 1 and  $l > 0$  decoded data is used additionally for derivation of  $ld[l]$ ).

When computing  $ld[numberOfDecomposedLayers-1], \dots, ld[0]$  all three arrays describing connectivity does not change their sizes, only values of their elements may change.

In the following is described how layers of a layered decomposition are derived by successively processing and manipulating mesh connectivity starting with the initial connectivity.

## K.2 Deriving layers $ld[l]$ , $l > 0$ using decoded data and connectivity ( $layeredDecompositionIsEncoded$ equals 1)

From the data decoded by the `FAMCLayeredDecompositionDecoder` class, simplification operations  $vvsop[l][c]$  (with  $l = numberOfDecomposedLayers-1, \dots, 1$  and  $c = 0, \dots, numberOfVerticesInLayer[l]$ ) are derived. Note that arrays  $vvsop[l]$ ,  $l > 0$  have the same sizes as arrays  $ld[l]$ ,  $l > 0$  will have after their derivation.

Layers  $ld[l]$  are derived successively for  $l = numberOfDecomposedLayers-1, \dots, 1$ . For this simplification operations  $vvsop[l][c]$  are applied for each  $l$  for  $c = 0, \dots, numberOfVerticesInLayer$ . Each time before applying a simplification operation the following data is stored:

- in  $ld[l][c]$  .to the vertex index to be removed  $vvsop[l][c].vertexIndex$  is stored, and
- in  $ld[l][c]$  .from from all vertex indices of the one-ring neighborhood of vertex  $vvsop[l][c].vertexIndex$  are stored.

After deriving a layer  $ld[l]$  all its elements are stored in reverse order. The following pseudo code illustrates the derivation process of  $ld[l]$ :

```
for (int l=numberOfDecomposedLayer-1; l>=1; --l)
  for (int c=0; c<numberOfVerticesInLayer[l]; ++c){
    ld[l][c].to = vvsop[l][c].vertexIndex;
    ld[l][c].from = getOneRingVertices(vvsop[l][c].vertexIndex, fv[][]);
    simplify(vvsop[l][c], fv[], isVertexDeleted[], isFaceDeleted[]);
  }
  reverse(ld[l]);
}
```

A description of functions `getOneRingVertices()` and `simplify()` can be found in K.5. Simplified connectivity obtained after derivation of  $ld[1]$  is used for obtaining  $ld[0]$ . This process is described below.

### K.3 Deriving layers $ld[l]$ , $l > 0$ using only connectivity (default option, `layeredDecompositionIsEncoded` equals 0)

The only difference of this derivation process compared to the derivation process described in previous section is that now simplification operation `vsop[][]` are not computed using decoded data and mesh connectivity, but they are computed from mesh connectivity only. In order to specify how layers  $ld[l]$  for  $l > 0$  are computed it is sufficient to describe how simplification operations `vsop[][]` are derived from connectivity; the rest of the procedure is similar with the one specified in the previous section.

Using initial connectivity as input, layers  $ld[l]$  are obtained in the order

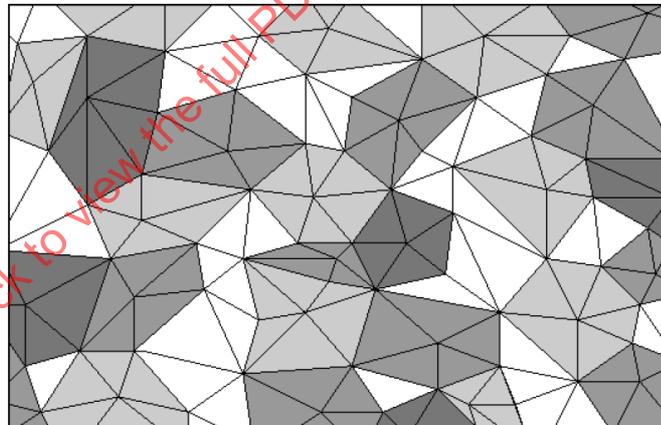
$l = \text{numberOfDecomposedLayers} - 1, \dots, 1$ .

Thereby, each layer  $ld[l]$  is derived in two stages:

- a) determination of a decomposition in patches `patch[l]`,
- b) derivation of simplification operations `vsop[l]` using `patch[l]` and simplification. Simplified connectivity is applied for obtaining  $ld[l-1]$ .

Simplified connectivity obtained after obtaining  $ld[1]$  is used for computing  $ld[0]$ . This derivation process is described in K.4.

In stage a) the mesh connectivity is decomposed in non-overlapping patches as illustrated in Figure K.1. All patches are described by an array of vertex indices `patch[l][c]` with  $c = 0, \dots, \text{numberOfPatchesInLayer}[l]$ . A patch `patch[l][c]` consists of all faces containing vertex index `patch[l][c]`. The number of these faces is called degree of vertex `patch[l][c]`.



**Figure K.1 — Decomposition of the mesh in patches. Gray shaded faces are part of patches, while white faces are not.**

The applied decomposition algorithm is a patch-based breath-first region-growing algorithm, which determines only patches with degree less or equal to 6. In the following the pseudo code of function `decompose(patch[])` is given, which describes the decomposition algorithm:

```
void decompose(int patch[]) {
    int seed;
    int vCurrent;
    FIFO fifo;
    int localVertices[];
    bool vertexTag1[] = isVertexDeleted[];
    bool vertexTag2[] = isVertexDeleted[];
    bool faceTag[] = isFaceDeleted[];
}
```

```

while ((seed=getNextSeed())!=-1){
    fifo.push(seed);
    while (!fifo.empty()){
        vCurrent=fifo.front();
        fifo.pop();
        if (isFreePatch(vCurrent)){
            int deg = degree(vCurrent);
            if (3<=deg && deg<=6){
                if (hasOrientableOneRingVertices(vCurrent, fv[][])){
                    conquerPatch(vCurrent);
                    patch.push_back(vCurrent);
                }
            }
        }
        if (vertexTag2[vCurrent]==false){
            vertexTag2[vCurrent]=true;
            localVertices=getOneRingVertices(vCurrent, fv[][]);
            for (int i=0; i<localVertices.size(); ++i){
                if (vertexTag2[localVertices[i]]==false){
                    fifo.push(localVertices[i]);
                }
            }
        }
    }
}
}
}

```

- o Variable fifo is a first-in first-out datastructure (FIFO), which stores vertex indices.
- o Function getNextSeed() returns the first vertexIndex for which vertexTag2[vertexIndex] is false. If all elements of vertexTag2[] are true -1 is returned.
- o Function isFreePatch(vCurrent) returns true iff vertexTag1[vCurrent] is false and for all faces f contating vCurrent faceTag[f] is false.
- o Function hasOrientableOneRingVertices(vCurrent, fv[][]) returns true iff all neighboring one-ring vertices of vertex vCurrent can be sorted in counter clock-wise order.

In stage b) an array vvsop[] of simplification operations is determined. Patches patch[][c] are traversed in order c=0,..., numberOfPatchesInLayer[] and a simplification mode mode[][c] is determined for each vertex index patch[][c]. The simplification mode is determined based on a cost function. Thereby that simplification mode is determined, which leads to the lowest cost. The cost function estimates the absolute deviation of degrees of vertices in the one-ring neighbourhood of vertex index patch[][c] from degree 6, after performing a simplification operation with a given simplification mode. Below pseudo code is given, which describes the determination of a simplification mode for a vertex index vertexIndex:

```

int determineSimplificationMode(int vertexIndex) {
    int mode = -1;
    int cost = -1;
    int costTmp = -1;
    ring[] = getOneRingVertices(vertexIndex, fv[][]);
    for (int k=0; k<ring.size(); ++k){
        costTmp = edgeCollapseCost(vertexIndex, k, fv[][]);
        if ((cost<0 && costTmp>=0) || (0<=costTmp && costTmp<cost)){
            cost = costTmp;
            mode = k;
        }
    }
    if (ring.size()==6){
        for (int k=6; k<8; ++k){
            costTmp = regularSimpificationCost(vertexIndex, k, fv[][]);
            if ((cost<0 && costTmp>=0) || (0<=costTmp && costTmp<cost)){
                cost = costTmp;
                mode = k;
            }
        }
    }
}

```

```

    }
  }
}
return mode;
}

```

The pseudo code of functions `edgeCollapseCost()` and `regularSimplificationCost()`, which calculate the cost of a simplification operation, is given below:

```

int edgeCollapseCost(SimplificationOperation op){
  int ring[] = getOneRingVertices(op.vertexIndex, fv[][]);
  if (ring.size()<3) return -1;
  if (ring.size()<=op.mode) return -1;
  int vertexIndexCollapse = ring[op.mode];
  if (isEdgeCollapseAllowed(op.vertexIndex, vertexIndexCollapse, fv[][]))
    return -1;
  adjustToIdx(ring[], vertexIndexCollapse);
  int deg[];
  for (int k=0; k<ring.size(); ++k){
    deg[k] = degree(ring[k]);
    deg[k]-=6;
  }
  deg[0] += degree(op.vertexIndex) - 4;
  deg[1] += -1;
  deg[deg.size()-1] += -1;
  return sumAbs(deg[]);
}

int regularSimplificationCost(SimplificationOperation op){
  if(op.mode!=6 && op.mode!=7) return -1;
  int ring[] = getOneRingVertices(op.vertexIndex, fv[][]);
  if (ring.size()!=6) return -1;
  int vertexIndexCollapse=ring[1];
  if (op.mode==6){
    vertexIndexCollapse = ring[0];
  }
  if (!isRegularSimplificationAllowed(op.vertexIndex, vertexIndexCollapse, fv[][]))
    return -1;
  adjustToIdx(ring[], vertexIndexCollapse);
  int deg[];
  for (int k=0; k<ring.size(); ++k){
    deg[k] = degree(ring[k]);
    deg[k]-=6;
    deg[k] += ((k%2)==0) ? 1 : -1;
  }
  return sumAbs(deg[]);
}

```

- Functions `isEdgeCollapseAllowed(...)` and `isRegularSimplificationAllowed(...)` are described in K.5.
- Function `adjustToIdx[ring[], vertexIndexCollapse]` rotates the positions of the elements in the array `ring[]` until element `vertexIndexCollapse` is at the first position of the array.
- Function `sumAbs(deg[])` calculates the sum of the absolute values of all valued stored in the array `deg[]`.

Finally simplification operations `op` with `op.vertexIndex=patch[l][c]` and `op.mode=mode[l][c]` are applied to mesh connectivity in order `c=0, ..., numberOfPatchesInLayer[l]`. For each `c`, if simplification operation `op` was successfully applied, `op` is stored in the array `vvsop[l]`.

Hence, after iteratively applying stage a) and b) for `l = numberOfDecomposedLayers-1, ..., 1` simplification operations `vvsop[l][c]` are completely specified.

## K.4 Deriving layer Id[0]

The base layer of the layered decomposition Id[0] is derived from simplified connectivity. Vertices of (simplified) connectivity are traversed in an order, which is specified by a face-based breath-first region-growing algorithm. Subsequently the obtained vertex order is used to derive vertex contexts Id[0][c] for  $c=0, \dots, \text{numberOfVerticesInLayer}[0]$ .

The pseudo code below describes the traversal algorithm. Function `traverse(int vertexIndices[])` determines an array `vertexIndices[]` of vertex indices in the order of traversal.

```
void traverse(int vertexIndices[]){
    int seed;
    int fCurrent;
    FIFO fifo;
    int localFaces[];
    bool faceTag[]=isFaceDeleted[];
    bool vertexTag[]=isVertexDeleted[];

    while ((seed=getNextSeed())!=-1){
        fifo.push(seed);
        while (!fifo.empty()){
            fCurrent=fifo.front();
            fifo.pop();
            if (faceTag[fCurrent]==false){
                faceTag[fCurrent]=true;
                localFaces=getNeighboringFaces(fCurrent, fv[][]);
                for (int i=0; i<(int)localFaces.size(); ++i){
                    if (faceTag[localFaces[i]]==false){
                        fifo.push(localFaces[i]);
                    }
                }
            }
            addVertices(fCurrent, vertexIndices[]);
        }
    }
    addRemainingVertices(vertexIndices[]);
}
```

- Function `getNextSeed()` returns the first face index `f` for which `faceTag[f]` is false. If all elements of `faceTag[]` are true -1 is returned.
- Function `getNeighboringFaces(fCurrent, fv[][])` returns face indices of all faces which have a common edge with face `fCurrent`.
- Function `addVertices(fCurrent, vertexIndices[])` determines all vertex indices `v` contained in face `fCurrent` for which `vertexTag[v]` is false, stores all determined vertex indices `v` in the array `vertexIndices[]`, and sets `vertexTag[v]` to true.
- Function `addRemainingVertices(vertexIndices[])` determines all vertex indices `v` for which `vertexTag[v]` is false, stores each determined `v` in the array `vertexIndices[]` and sets `vertexTag[v]` to true.

In order to derive vertex contexts Id[0][c] for each  $c=0, \dots, \text{numberOfVerticesInLayer}[0]$  the obtained array of vertex indices `vertexIndex[]` is exploited. For this vertex indices `vertexIndex[c]` are traversed for  $c=0, \dots, \text{numberOfVerticesInLayer}[0]$ . Thereby for each `c`

- the value of `Id[0][c].to` is set to `vertexIndices[c]`,
- in the array `Id[0][c].from` all vertex indices `v` are stored, which are located in the one-ring neighbourhood of vertex `vertexIndices[c]` and which were visited previously,
- The value of `isVisited[vertexIndex[c]]` is set to true.

The pseudo code below describes the derivation algorithm for `ld[0]`:

```
vector<bool> isVisited(numberOfVerticesInLayer[0], false);
for (int c=0; c<numberOfVerticesInLayer[0]; ++c){
    ld[0][c].to = vertexIndices[c];
    int ring[] = getOneRingVertices(vertexIndices[c]);
    for (int k=0; k<ring.size(); ++k){
        if (isVisited[ring[k]]){
            ld[0][c].from.push_back(ring[k]);
        }
    }
    isVisited[vertexIndices[c]]=true;
}
```

## K.5 Mesh simplification

The application of a simplification operation *op* to mesh connectivity is realized as illustrated with the pseudo code below. Depending on the number of one-ring vertices and mode, one of two types of simplification operations is applied. For the case that the number of one-ring vertices of vertex *op.vertexIndex* is 6 and *op.mode* is 6 or 7 a regular simplification operation is applied. Otherwise, if *op.mode* is lower than the number of one-ring vertices, an edge collapse operation is applied.

```
bool simplify(SimplificationOperation op){
    int ring[] = getOneRingVertices(op.vertexIndex);
    if (ring.size()==6) {
        if (op.mode==6){
            int vertexIndexCollapse = ring[1];
            if (isRegularSimplificationAllowed(op.vertexIndex, vertexIndexCollapse))
                return regularSimplification(op.vertexIndex, vertexIndexCollapse);
        }
        if (op.mode==7){
            int vertexIndexCollapse = ring[0];
            if (isRegularSimplificationAllowed(op.vertexIndex, vertexIndexCollapse))
                return regularSimplification(op.vertexIndex, vertexIndexCollapse);
        }
    }
    if (op.mode<ring.size()) {
        int vertexIndexCollapse = ring[op.mode];
        if (isEdgeCollapseAllowed(op.vertexIndex, vertexIndexCollapse))
            return edgeCollapse(op.vertexIndex, vertexIndexCollapse);
    }
    return false;
}
```

Function `getOneRingVertices(op.vertexIndex)` determines all vertices which are located in the one-ring neighbourhood of `vertexIndex`. If these vertices are orientable in counter clock-wise order, an array of vertex indices in this order is returned. Otherwise vertex indices are sorted in increasing order, and an array of vertex indices in increasing order is returned. In the following we show the pseudo code of this function:

```
int[] getOneRingVertices(int vertexIndex) {
    if (hasOrientableOneRingVertices(vertexIndex)){
        return getOneRingVerticesCCW(vertexIndex);
    }
    return getOneRingVerticesInc(vertexIndex);
}
```

Before applying an edge collapse or a regular simplification operation, respectively, it is checked if it is allowed to apply this operation. An edge collapse operations is allowed, if `vertexIndex` and `vertexIndexCollapse` are neighbouring vertices:

```
bool isEdgeCollapseAllowed(int vertexIndex, int vertexIndexCollapse) {
    if (!areNeighboringVertices(vertexIndex, vertexIndexCollapse)) return false;
    return true;
}
```

A regular simplification operation is allowed, if vertexIndex and vertexIndexCollapse are neighbouring vertices, if the number of vertices in the one-ring neighbourhood of vertexIndex is 6, and if these vertices are orientable in counter clock-wise order:

```
bool isRegularSimplificationAllowed(int vertexIndex, int vertexIndexCollapse){
    if (!isEdgeCollapseAllowed(vertexIndex, vertexIndexCollapse)) return false;
    int ring[] = getOneRingVertices(_vertexIndex);
    if (ring.size()!=6) return false;
    if (!isClosedOrientablePatch(vertexIndex)) return false;
    return true;
}
```

In the following is described how an edge collapse operation is performed. Figure K.2 illustrates this process. First all faces that contain edge (vertexIndex, vertexIndexCollapse) are marked as deleted in the array isFaceDeleted[] (shaded faces in Figure K.2 ). Subsequently, for each of the remaining patch faces, in the array fv[][] the value of vertexIndex is replaced with vertexIndexCollapse. Finally isVertexDeleted[vertexIndex] is set to true.



Figure K.2 — Illustration of an edge collapse operation.

A regular simplification operation is implemented as an extension of an edge collapse operation. First an edge collapse operation is performed, which collapses vertex index vertexIndex into vertexIndexCollapse. Subsequently, the common edge of faces f0 and f1 (see Figure K.3) is flipped and arrays fv[f0] and fv[f1] are updated accordingly.

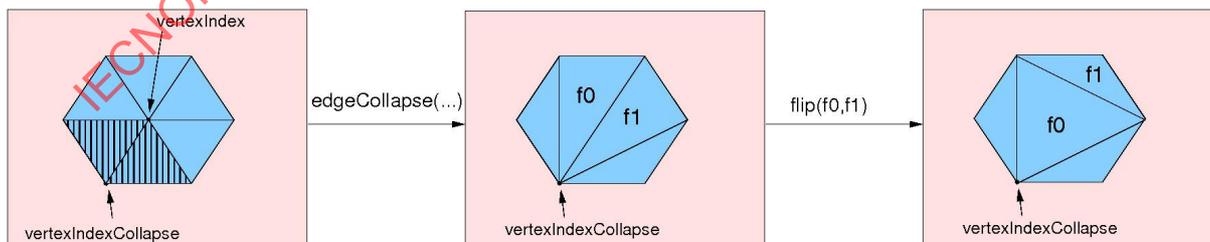


Figure K.3 — Illustration of a regular simplification operation.

## Annex L (normative)

### Reconstruction of values from decoded prediction errors with LD technique

For each quantized prediction error (resCoords[frameNumberDis][layerNumber][c], resNormals[frameNumberDis][layerNumber][c], resColors[frameNumberDis][layerNumber][c], or resOtherAttributes[frameNumberDis][layerNumber][c], respectively) in a frame with display frame number equals to frameNumberDis, a layer with layer number layerNumber, and with c in {0, ..., numberOfVerticesInLayer[layerNumber]} a corresponding reconstructed value is obtained. The process is based on the following data:

- coordsQuantizationStepLD (resp. normalsQuantizationStepLD, colorsQuantizationStepLD, and otherAttributesQuantizationStepLD)
- frameType,
- frameNumberDis, and optionally refFrameNumberOffsetDis0 and refFrameNumberOffsetDis1,
- ld[layerNumber][c],
- coordsPredType[frameNumberDis][layerNumber][c] (resp. normalsPredType[], colorsPredType[], and otherAttributesPredType[])
- res[frameNumberDis][layerNumber][c],
- previously reconstructed values.

Below we describe the reconstruction process for coordinates. The reconstruction process for normals, colors and other attributes is similar.

In following we use the following notations:

$$\Delta := \text{quantizationStepLD}$$

$$T^{\text{frame}} = \text{frameType}$$

$$f := \text{frameNumberDis},$$

$$f_{r1} := \text{refFrameNumberDis0},$$

$$f_{r2} := \text{refFrameNumberDis1},$$

$$v := \text{ld}[\text{layerNumber}][c],$$

$$v_0 := \text{ld}[\text{layerNumber}][c], \text{from}[0],$$

$$v_1 := \text{ld}[\text{layerNumber}][c], \text{from}[1],$$

$$N(v) := \text{ld}[\text{layerNumber}][c], \text{from},$$

$$T^{\text{pred}}(f, v) := \text{predType}[\text{frameNumberDis}][\text{layerNumber}][c],$$

$$\text{res}(f, v) := \text{res}[\text{frameNumberDis}][\text{layerNumber}][c],$$

Let  $\bar{p}_v^f$  be the reconstructed 3D coordinate corresponding to vertex  $v$  in frame  $f$ .