INTERNATIONAL STANDARD

ISO/IEC 14496-16

Second edition
2006-12-15

AMENDMENT 2
2009-02-15

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)

Reference number
ISO/IEC 14496-16:2006/Amd.2:2009(E)

© ISO/IEC 2009

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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.



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
Web www.iso.org

Published in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2009 – All rights reserved

Not for Resale

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.

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

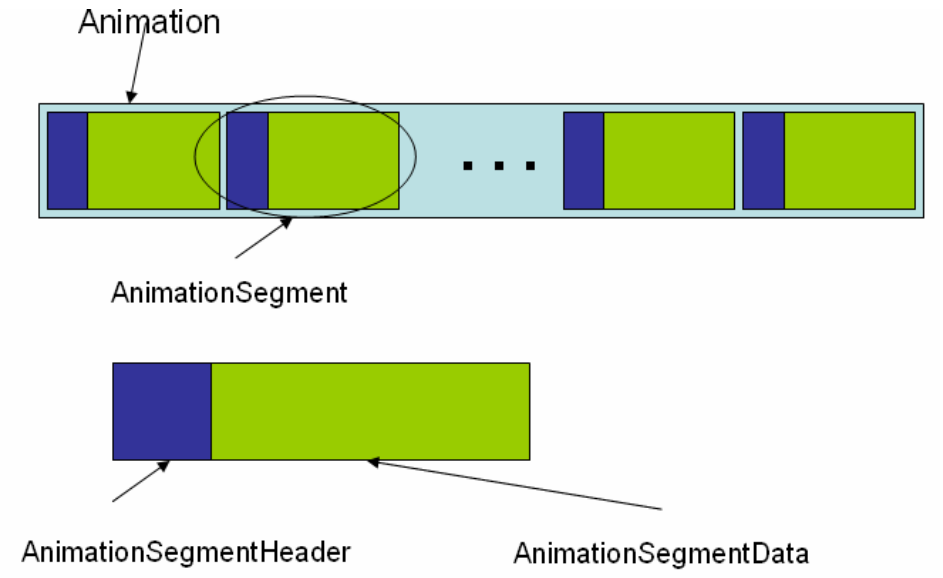

Figure AMD2.1 — FAMC bitstream structure.

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:

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- 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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

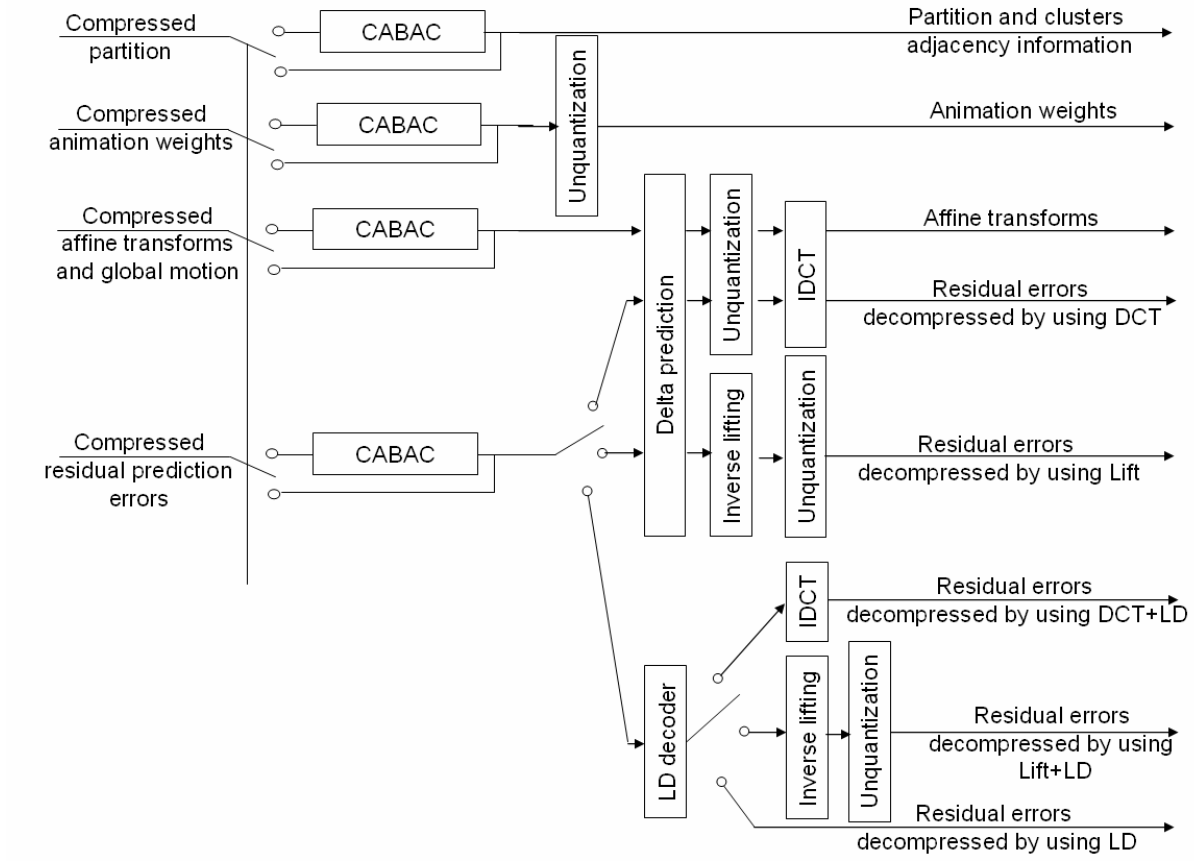

Figure AMD2.2 — FAMC decoding process.

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.

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.10.4 FAMCAnimationSegment class

5.10.4.1 Syntax

```
class FAMCAnimationSegment {
   FAMCAnimationSegmentHeader header;
   FAMCAnimationSegmentData data;
}
```

5.10.4.2 Semantics

FAMCAnimationSegmentHeader: contains the header buffer.

FAMCAnimationSegmentData: contains the data buffer.

5.10.5 FAMCAnimationSegmentHeader class

5.10.5.1 Syntax

```
class  FAMCAnimationSegmentHeader {
   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 **FAMCAnimationSegmentStartCode**.

staticMeshDecodingType: a 8-bit unsigned integer indicating if the static mesh is encoded whithin 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.

firstFrameDecodingType value	First frame decoding type
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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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;
}
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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;
   FAMCAnimationPartitionDecoder   partition;
   FAMCAffineTrasnformsDecoder     affineTransforms;
   FAMCAnimationWeightsDecoder     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.
- **FAMCAnimationPartition** 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.
- **FAMCAnimationWeights** 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;
         FamcCabacVx3Decoder2 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.

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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++;
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// 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;
```

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            }
         }
      }
   }

   // 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{
```


© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            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 x 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][c]][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);
}
```

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 FAMCAnimationPartition 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 occurence = 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));
      }
      occurenceMinusOne = cabac.unary_exp_golomb_decode(cabac._dep, cabac._ctx, 2);
      for (int i =0; i < occurenceMinusOne+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.

occurenceMinusOne: the number minus one of consecutive verticexIndex 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);
}
```


© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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_i^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 compute the sequences of (A_t^k) the decoder simply apply the following linear equation:

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

5.10.14 FAMCAnimationWeightsDecoder class

5.10.14.1 Syntax

```
class FAMCAnimationWeightsDecoder {
   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
```

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   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 dimention 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:

numberOfBits = (int) (log((double) 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:

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

where ω_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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The optimal weight vector $\omega^v = \left(\omega_k^v\right)_{k\in\{1,\ldots,\,numberOfClusters\}}$ is computed at the encoder side and sent to the decoder. The ω_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 retrive the values of weights, the quantization process needs to be reversed.

5.10.15 FAMCAllResidualErrors

5.10.15.1 Syntax

```
class FAMCAllResidualErrors {
       switch (transformType) {
         case 0 : // Lifting
         case 1 : // DCT
           if (animatedFields & 1)
              FAMCInfoTableDecoder coordResidualErrorsInfo;
           if (animatedFields & 2)
              FAMCInfoTableDecoder normalResidualErrorsInfo;
           if (animatedFields & 4)
              FAMCInfoTableDecoder colorResidualErrorsInfo;
           if (animatedFields & 8)
              FAMCInfoTableDecoder 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==FAMCAnimationSegmentStartCode);
          break;
         case 2: // LD
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 transformeType is DCT.

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

$$(\varepsilon_t^v)_{\substack{v\in\{1,...,numberOfCoord\}\\ t\in\{2,...,numberOfCoordKeys\}}}$$

expressed in homogeneous coordinates. The decoder computes the position χ_t^v of a vertex ***v*** at frame ***t*** by applying the following formula:

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

where χ_1^v is the position of vertex ***v*** at the first frame, A_t^k is the affine transform associated with the cluster ***k*** at frame ***t*** expressed in homogeneous coordinates, ω_k^v: the weight of cluster ***k*** at vertex ***v***, γ_t: the global motion of frame ***t***, ε_t^v: coordinate residual errors of vertex ***v*** at frame ***t*** .

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

$$(n_t^v)_{\substack{v\in\{1,...,numberOfCoord\}\\ t\in\{2,...,numberOfCoordKeys\}}}$$

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

$N_t^v = N_1^v + n_t^v$ if normalsPredictionStrategy=0

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

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

where

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

N_1^v is the normal of vertex ***v*** at the first frame and n_t^v is the normal residual errors of vertex ***v*** at frame ***t*** .

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

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

© ISO/IEC 2009 All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 =
         ((otherAttributsPredictionModeLD == 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)
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            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

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 ld[][] 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.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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
coordsPredType[frameNumberDis][layerNumber][c]=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 coordsPredBits[frameNumberDis][layerNumber][c] , which is used to derive definite prediction types.

If coordsPredictionModeLD=3 then
coordsPredType[frameNumberDis][layerNumber][c] =1 if coordsPredBits[frameNumberDis][layerNumber][c] ==1 and
coordsPredType[frameNumberDis][layerNumber][c] =2 if coordPredBits[frameNumberDis][layerNumber][c] ==0.

If coordsPredictionModeLD=4 then
coordsPredType[frameNumberDis][layerNumber][c] = coordPredBits[frameNumberDis][layerNumber][c] .

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

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 indicats the assignement 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:

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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).

frameNumberDis=frameNumberPrevDis+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:

refFrameNumberDec0=frameNumberDec – 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:

refFrameNumberDec1=frameNumberDec – 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:

frameNumberDec2DisList[frameNumberDec]=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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 );
   }
}
```

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 x 3 of bits indicating the non-zero values.

negative: an arithmetic encoded array of size V x 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 decodde for decoding of absolute values.

values: an arithmetic encoded array of size V x 3 of integers values.

predModeInitProb: 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 x 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, Gemany
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:

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

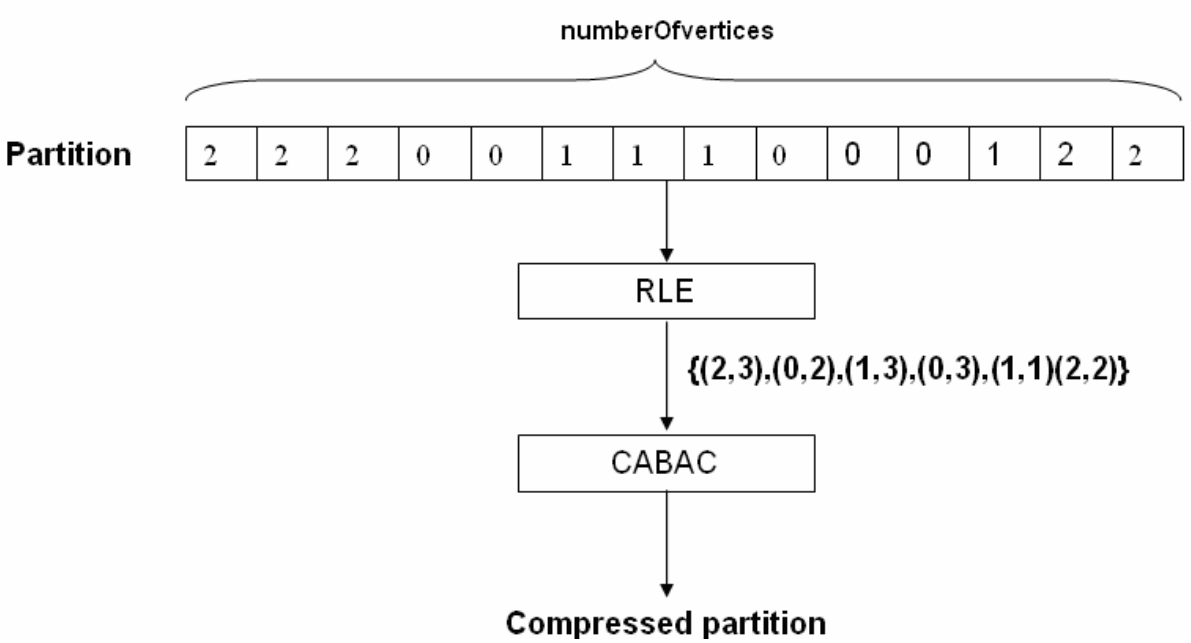

Figure I.1— RLE encoding of the vertices partition: example with 14 vertices and 3 clusters.

© ISO/IEC 2009 All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 $\omega_k^v = \delta_{k,l}, \forall l \in = \{1,...,numberOfClusters\}$ ($\delta_{k,l}$ 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 $(N(k))_{k\in\{1,...,numberOfClusters\}}$ 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.

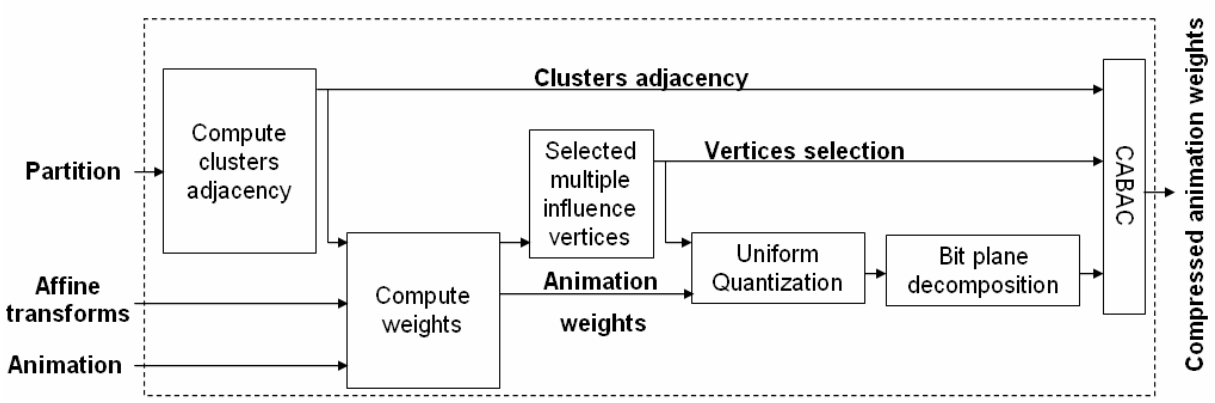

Figure J.1 — Animation weights encoding.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex K
(normative)

Layered decomposition

K.1 Obtaining the layered decomposition

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

ld[l]

with l=0,…, 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; ++k){
      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].from[k]]=true;
      }
   }
   return true;
}
```

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

ld[numberOfDecomposedLayers-1],…,ld[0].

There exist two ways to derive layers ld[numberOfDecomposedLayers-1],…,ld[1]:

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

© ISO/IEC 2009 All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 numberOfFacesx3,
- 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],…,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,…,1 and c=0,…, 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,…,1. For this simplification operrations vvsop[l][c] are applied for each l for c=0,…,numberOfVerticesInLayer. Each time before applying a simplification operation the follwing 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 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[][]:

```
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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 vvsop[][] 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 vvsop[][] 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 = numberOfDecomposedLayers-1,…,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 vvsop[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,…,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].

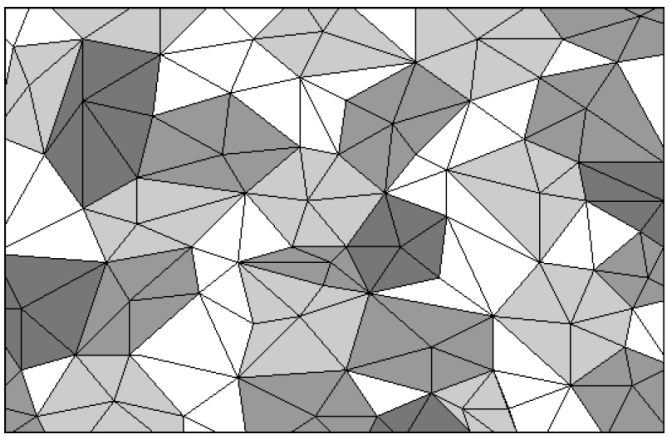

Figure K.1 — Decomposition of the mesh in patches. Gray shaded faces are part of patches, while white faces are not.

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[];
```

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   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]);
               }
            }
         }
      }
   }
}
```

- Variable fifo is a first-in first-out datastructure (FIFO), which stores vertex indices.
- Function getNextSeed() returns the first vertexIndex for which vertexTag2[vertexIndex] is false. If all elements of vertexTag2[] are true -1 is returned.
- Function isFreePatch(vCurrent) returns true iff vertexTag1[vCurrent] is false and for all faces f contating vCurrent faceTag[f] is false.
- 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[l] of simplification operations is determined. Patches patch[l][c] are traversed in order c=0,…, numberOfPatchesInLayer[l] and a simplification mode mode[l][c] is determined for each vertex index patch[l][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[l][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;
```


© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        }
      }
   }
   return mode;
}
```

The pseudo code of functions edgeCollapseCost() and regularSimpificationCost(), 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 regularSimpificationCost(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 interativelly applying stage a) and b) for l = numberOfDecomposedLayers-1,…,1 simplification operations vvsop[][] are completelly specified.

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

K.4 Deriving layer ld[0]

The base layer of the layered decomposition ld[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 ld[0][c] for c=0,…,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 traveral.

```
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 ld[0][c] for each c=0,…,numberOfVerticesInLayer[0] the obtained array of vertex indices vertexIndex[] is exploited. For this vertex indices vertexIndex[c] are traversed for c=0,…,numberOfVerticesInLayer[0]. Thereby for each c

- the value of ld[0][c].to is set to vertexIndices[c],

- in the array ld[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.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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, respecively, it is checked if it is allowed to apply this operation. An edge collapse operations is allowed, if vertexIndex and vertexIndexCollapse are neighbouring vertices:

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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.

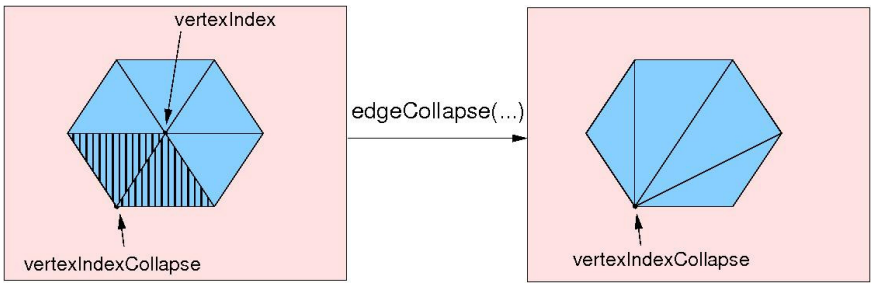

Figure K.2 — Illustration of an edge collapse operation.

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.

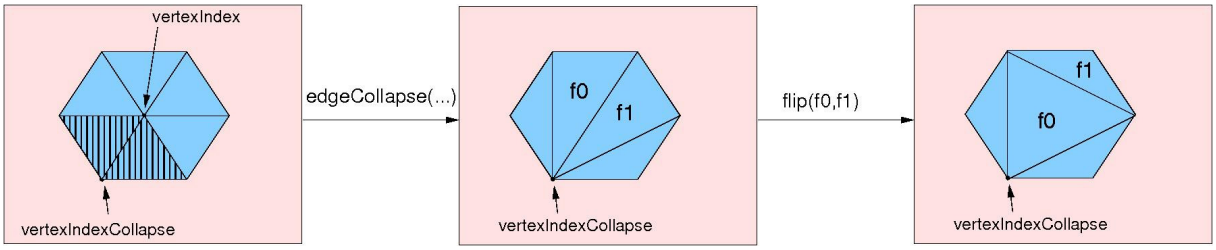

Figure K.3 — Illustration of a regular simplification operation.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 resOtherAttirbutes[frameNumberDis][layerNumber][c], respectivelly) 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:

$$\Delta := quantizeationStepLD$$
$$T^{frame} = frameType$$
$$f := frameNumberDis,$$
$$f_{r1} := refFrameNumberDis0,$$
$$f_{r2} := refFrameNumberDis1,$$
$$v := ld[layerNumber][c].to,$$
$$v_0 := ld[layerNumber][c].from[0],$$
$$v_1 := ld[layerNumber][c].from[1],$$
$$N(v) := ld[layerNumber][c].from,$$
$$T^{pred}(f,v) := predType[frameNumberDis][layerNumber][c],$$
$$res(f,v) := res[frameNumberDis][layerNumber][c],$$

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

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Each real valued 3D coordinate, $\overline{p}_v^f \in R^3$ is computed as follows:

$$\overline{p}_v^f = pred(f, v, f_{r1}, f_{r2}, T^{frame}, T^{pred}) + \Delta \cdot res(f, v),$$

with

$$pred(f, v, f_{r1}, f_{r2}, T^{frame}, T^{pred}) := \begin{cases} predDelta(f, v, f_{r1}, f_{r2}, T^{frame}) & : T^{pred}(f, v) = 0 \\ predLin(f, v, f_{r1}, f_{r2}, T^{frame}) & : T^{pred}(f, v) = 1 \\ predNonLin(f, v, f_{r1}, f_{r2}, T^{frame}) & : T^{pred}(f, v) = 2 \end{cases}$$

All three predictors $predDelta$, $predLin$ and $predNonLin$ support 3 different prediction modes, depending on the frame type, which is specified by variable T^{frame} : I-frame, P-frame, and B-frame:

$$predDelta(f, v, f_{r1}, f_{r2}, T^{frame}) := \begin{cases} predDelta^I(f, v) & : T^{frame} = 0 \\ predDelta^P(f, v, f_{r1}) & : T^{frame} = 1 \\ predDelta^B(f, v, f_{r1}, f_{r2}) & : T^{frame} = 2 \end{cases}$$

$$predLin(f, v, f_{r1}, f_{r2}, T^{frame}) := \begin{cases} predLin^I(f, v) & : T^{frame} = 0 \\ predLin^P(f, v, f_{r1}) & : T^{frame} = 1, \\ predLin^B(f, v, f_{r1}, f_{r2}) & : T^{frame} = 2 \end{cases}$$

$$predNonLin(f, v, f_{r1}, f_{r2}, T^{frame}) := \begin{cases} predNonLin^I(f, v) & : T^{frame} = 0 \\ predNonLin^P(f, v, f_{r1}) & : T^{frame} = 1. \\ predNonLin^B(f, v, f_{r1}, f_{r2}) & : T^{frame} = 2 \end{cases}$$

- **frame prediction**

For I-frames, $predDelta$ predicts always a zero vector:

$$predDelta^I(f, v) := \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Predictors $predLin$ and $predNonLin$ perform the same prediction for I-frames, both are predicting the barycenter :

$$predLin^I(f, v) := predNonLin^I(f, v) := bary(f, v) := \frac{1}{|N(v)|} \sum_{v' \in N(v)} \overline{p}_{v'}^f$$

$|N(v)|$: number of elements in $N(v)$.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **P-frame prediction**

For P-frames, $predDelta$ predicts as follows:

$$predDelta^P(f,v,f_{r1}) := \overline{p}_v^{f_{r1}}.$$

Predictors $predLin$ and $predNonLin$ perform for P-frames a prediction of the same type:

$$predLin^P(f,v,f_{r1}) := bary(f,v) + A_{lin}(\overline{p}_v^{f_{r1}} - bary(f_{r1},v)),$$

$$predNonLin^P(f,v,f_{r1}) := bary(f,v) + A_{nonlin}(\overline{p}_v^{f_{r1}} - bary(f_{r1},v)).$$

A_{lin}, A_{nonlin}: 3x3 matrices.

The only difference between both predictors is in the matrices A_{lin}, A_{nonlin}. For $predLin^P$ we use $A_{lin} = I$, the identity matrix.

For $predNonLin^P$ a A_{nonlin} is calculated based on reconstructed 3D points.

A_{nonlin} is determined by calculating two orthogonal local coordinate frames based on already reconstructed 3D points, one in frame f and one in frame f_{r1}.

Calculation of a local coordinate frame is performed based on 3 3D points: 2 reconstructed 3D points assigned to vertex indices v_0 and v_1 and the barycenter corresponding to the one-ring vertices in $N(v)$.

First an orthogonal local coordinate frame is calculated in frame f as follows:
$\vec{x}_f := \frac{\overline{p}_{v_0}^f - bary(f,v)}{\left\|\overline{p}_{v_0}^f - bary(f,v)\right\|}$, $\vec{y}_f^{tmp} := \frac{\overline{p}_{v_1}^f - bary(f,v)}{\left\|\overline{p}_{v_1}^f - bary(f,v)\right\|}$, $\vec{z}_f = \frac{\vec{x}_f \times \vec{y}_f^{tmp}}{\left\|\vec{x}_f \times \vec{y}_f^{tmp}\right\|}$ and $\vec{y}_f = \vec{z}_f \times \vec{x}_f$. Let denote this coordinate frame with
$M_f := (\vec{x}_f, \vec{y}_f, \vec{z}_f).$
Thereafter, a corresponding local coordinate frame in calculated in frame f_{r1}, denoted with
$M_{f_{r1}} := (\vec{x}_{f_{r1}}, \vec{y}_{f_{r1}}, \vec{z}_{f_{r1}})$.

Finally we obtain $A_{nonlin} := M_f M_{f_{r1}}{}^T$, which specifies predictor $predNonLin^P(f,v,f_{r1})$.

In the case that

- there are less or equal to 3 one ring vertices (i.e. $|N(v)| \le 3$) or
- 3D points, which are used for calculation of $M_{f_{r1}}$ or M_f , respectively, are linearly dependent,

we define $A_{nonlin} := A_{lin} := I$.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **B-frame prediction**

B-frame prediction is performed by calculating the average of two P-frame predictions:

$$predDelta^{B}(f,v,f_{r1},f_{r2}) := \frac{1}{2}(predDelta^{P}(f,v,f_{r1}) + predDelta^{P}(f,v,f_{r2})$$

$$predLin^{B}(f,v,f_{r1},f_{r2}) := \frac{1}{2}(predLin^{P}(f,v,f_{r1}) + predLin^{P}(f,v,f_{r2}),$$

$$predNonLin^{B}(f,v,f_{r1},f_{r2}) := \frac{1}{2}(predNonLin^{P}(f,v,f_{r1}) + predNonLin^{P}(f,v,f_{r2},)).$$

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex M
(normative)

CABAC definitions, basic functions, and binarizations

M.1 CABAC definitions

M.1.1 Definition of the term `CabacContext`

A variable of type `CabacContext` consists of the unsigned integer variables `pStateIdx` and `valMPS`.

`cabac._ctx` is a global variable of type `CabacContext`.

M.1.2 Definition of the term `DecodingEnvironment`

A variable of type `DecodingEnvironment` consists of the unsigned integer variables `codIRange` and `codIOffset`.

`cabac._dep` is a global variable of type `DecodingEnvironment`.

M.2 CABAC basic functions

M.2.1 Definition of the function `cabac.biari_init_context( ctx, preCtxState )`

The variable `ctx` is of type `CabacContext`.

The variable `preCtxState` is of type `unsigned int`.

Whenever the function `cabac.biari_init_context` is called, `ctx` is modified as follows:

```
if( preCtxState  <=  63 ) {
   ctx.pStateIdx = 63 - preCtxState
   ctx.valMPS = 0
} else {
   ctx.pStateIdx = preCtxState - 64
   ctx.valMPS = 1
}
```

M.2.2 Definition of the function `cabac.arideco_start_decoding( dec_env )`

The variable `dec_env` is of type `DecodingEnvironment`.

Whenever the function `cabac.arideco_start_decoding` is called, `dec_env` is modified as follows:

```
dec_env.codIRange = 510
dec_env.codIOffset = bit( 9 )
```

M.2.3 Definition of the function `cabac.biari_decode_symbol( dec_env, ctx )`

The variable `dec_env` is of type `DecodingEnvironment`.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2009 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The variable `ctx` is of type `CabacContext`.

Whenever the function `cabac.biari_decode_symbol` is called, `dec_env` and `ctx` are modified by the following ordered steps:

1. An unsigned integer variable `codIRangeLPS` is derived as follows.

   – Given the current value of `dec_env.codIRange`, the variable `qCodIRangeIdx` is derived by `qCodIRangeIdx =( dec_env.codIRange >> 6 ) & 0x03`

   – Given `qCodIRangeIdx` and `ctx.pStateIdx`, the value of the variable `rangeTabLPS` as specified in Table M.1 is assigned to `codIRangeLPS`:

   ```
   codIRangeLPS = rangeTabLPS[ ctx.pStateIdx ][ qCodIRangeIdx ]
   ```

2. The variable `dec_env.codIRange` is set equal to `dec_env.codIRange - codIRangeLPS` and the following applies.

   – If `dec_env.codIOffset` is greater than or equal to `dec_env.codIRange`, the variable `binVal` is set equal to `1 - ctx.valMPS`, `dec_env.codIOffset` is decremented by `dec_env.codIRange`, and `dec_env.codIRange` is set equal to `codIRangeLPS`.

   – Otherwise, the variable `binVal` is set equal to `ctx.valMPS`.

3. Depending on the decoded value `binVal`, the update of the two variables `ctx.pStateIdx` and `ctx.valMPS` is derived as follows:

   ```
   if( binVal == ctx.valMPS )
       ctx.pStateIdx = transIdxMPS( ctx.pStateIdx )
   else {
     if( ctx.pStateIdx == 0 )
        ctx.valMPS = 1 - ctx.valMPS
     ctx.pStateIdx = transIdxLPS( ctx.pStateIdx )
   }
   ```

   Table M.2 specifies the transition rules `transIdxMPS( )` and `transIdxLPS( )`.

4. The renormalization process as specified in M.2.3.1 is performed with `dec_env` as input

5. The decoded value `binVal` is returned to the process invoking this subclause.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table M.1 — Specification of rangeTabLPS depending on pStateIdx and qCodIRangeIdx

pStateIdx	qCodIRangeIdx				pStateIdx	qCodIRangeIdx			
	0	1	2	3		0	1	2	3
0	128	176	208	240	**32**	27	33	39	45
1	128	167	197	227	**33**	26	31	37	43
2	128	158	187	216	**34**	24	30	35	41
3	123	150	178	205	**35**	23	28	33	39
4	116	142	169	195	**36**	22	27	32	37
5	111	135	160	185	**37**	21	26	30	35
6	105	128	152	175	**38**	20	24	29	33
7	100	122	144	166	**39**	19	23	27	31
8	95	116	137	158	**40**	18	22	26	30
9	90	110	130	150	**41**	17	21	25	28
10	85	104	123	142	**42**	16	20	23	27
11	81	99	117	135	**43**	15	19	22	25
12	77	94	111	128	**44**	14	18	21	24
13	73	89	105	122	**45**	14	17	20	23
14	69	85	100	116	**46**	13	16	19	22
15	66	80	95	110	**47**	12	15	18	21
16	62	76	90	104	**48**	12	14	17	20
17	59	72	86	99	**49**	11	14	16	19
18	56	69	81	94	**50**	11	13	15	18
19	53	65	77	89	**51**	10	12	15	17
20	51	62	73	85	**52**	10	12	14	16
21	48	59	69	80	**53**	9	11	13	15
22	46	56	66	76	**54**	9	11	12	14
23	43	53	63	72	**55**	8	10	12	14
24	41	50	59	69	**56**	8	9	11	13
25	39	48	56	65	**57**	7	9	11	12
26	37	45	54	62	**58**	7	9	10	12
27	35	43	51	59	**59**	7	8	10	11
28	33	41	48	56	**60**	6	8	9	11
29	32	39	46	53	**61**	6	7	9	10
30	30	37	43	50	**62**	6	7	8	9
31	29	35	41	48	**63**	2	2	2	2

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table M.2 — State transition table

pStateIdx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
transIdxLPS	0	0	1	2	2	4	4	5	6	7	8	9	9	11	11	12
transIdxMPS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pStateIdx	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
transIdxLPS	13	13	15	15	16	16	18	18	19	19	21	21	22	22	23	24
transIdxMPS	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
pStateIdx	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
transIdxLPS	24	25	26	26	27	27	28	29	29	30	30	30	31	32	32	33
transIdxMPS	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
pStateIdx	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
transIdxLPS	33	33	34	34	35	35	35	36	36	36	37	37	37	38	38	63
transIdxMPS	49	50	51	52	53	54	55	56	57	58	59	60	61	62	62	63

M.2.3.1 Renormalization process in the arithmetic decoding engine

Input to this process is a variable `dec_env` of type `DecodingEnvironment`.

The variable `dec_env` is modified as follows:

```
while( dec_env.codIRange <= 256 ) {
   dec_env.codIRange <<= 1
   dec_env.codIOffset = ( dec_env.codIOffset << 1 ) | bit( 1 )
}
```

The bitstream shall not contain data that results in a value of `dec_env.codIOffset` being greater than or equal to `dec_env.codIRange` upon completion of this process.

M.2.4 Definition of the function `cabac.biari_decode_symbol_eq_prob( dec_env )`

The variable `dec_env` is of type `DecodingEnvironment`.

Whenever the function `cabac.biari_decode_symbol_eq_prob` is called, `dec_env` is modified by the following ordered steps:

1. The value of `dec_env.codIOffset` is doubled, i.e., left-shifted by 1 and a single bit is shifted into `dec_env.codIOffset` by using `bit( 1 )`.

2. The value of `dec_env.codIOffset` is compared to the value of dec`_env.codIRange` and further steps are specified as follows.

   – If `dec_env.codIOffset` is greater than or equal to `dec_env.codIRange`, the variable `binVal` is set equal to 1 and `dec_env.codIOffset` is decremented by `dec_env.codIRange`.

   – Otherwise (`dec_env.codIOffset` is less than `dec_env.codIRange`), the variable `binVal` is set equal to 0.

3. The decoded value `binVal` is returned to the process invoking this subclause.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The bitstream shall not contain data that results in a value of `dec_env.codIOffset` being greater than or equal to `dec_env.codIRange` upon completion of this process.

M.2.5 Definition of the function `cabac.biari_decode_final( dec_env )`

The variable `dec_env` is of type `DecodingEnvironment`.

Whenever the function `cabac.biari_decode_final` is called, `dec_env` is modified by the following ordered steps:

1. The value of `dec_env.codIRange` is decremented by 2.

2. The value of `dec_env.codIOffset` is compared to the value of `dec_env.codIRange` and further steps are specified as follows.

   – If `dec_env.codIOffset` is greater than or equal to `dec_env.codIRange`, the variable `binVal` is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated.

   – Otherwise (`dec_env.codIOffset` is less than `dec_env.codIRange`), the variable `binVal` is set equal to 0 and renormalization is performed as specified in M.2.3.1.

3. The decoded value `binVal` is returned to the process invoking this subclause.

M.3 CABAC binarization functionss

CABAC binarization functions utilize CABAC basic functions only.

M.3.1 Definition of the function `cabac.exp_golomb_decode_eq_prob( dec_env, k )`

The variable `dec_env` is of type `DecodingEnvironment`.

The variable `k` is of type `unsigned int`.

Whenever the function `cabac.exp_golomb_decode_eq_prob` is called the following applies:

```
unsigned int l;
int symbol = 0;
int binary_symbol = 0;

do
{
   l = cabac.biari_decode_symbol_eq_prob( dec_env );
   if (l==1) {
      symbol += (1<<k);
      k++;
   }
}
while (l!=0);

while (k--)
   if( cabac.biari_decode_symbol_eq_prob( dec_env ) == 1 )
      binary_symbol |= (1<<k);
```

The sum of values `symbol` and `binary_symbol` is returned to the process invoking this subclause.

M.3.2 Definition of the function `cabac.unary_exp_golomb_decode( dec_env, ctx, exp_start )`

The variable `dec_env` is of type `DecodingEnvironment`.

© ISO/IEC 2009 All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The variable `ctx` is of type `CabacContext`.

The variable `exp_start` is of type `unsigned int`.

Whenever the function `cabac.unary_exp_golomb_decode` is called the following applies:

```
unsigned int l,k;
unsigned int symbol;

symbol = cabac.biari_decode_symbol( dec_env, ctx );

if (symbol!=0) {
  k=1;
  do
  {
    l = cabac.biari_decode_symbol(dec_env, ctx);
    symbol++;
    k++;
  }
  while( l != 0 && k != exp_start );
  if( l != 0 )
    symbol += cabac.exp_golomb_decode_eq_prob( dec_env, 0 ) + 1;
}
```

The value `symbol` is returned to the process invoking this subclause.

© ISO/IEC 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.040

Price based on 46 pages

© ISO 2009 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale