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 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

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.

Information technology — Coding of audio-visual objects —

Part 16:
Animation Framework eXtension (AFX)

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

After 5.9, add the following new subclause:

5.10 Frame-based Animated Mesh Compression (FAMC) stream

5.10.1 Overview

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

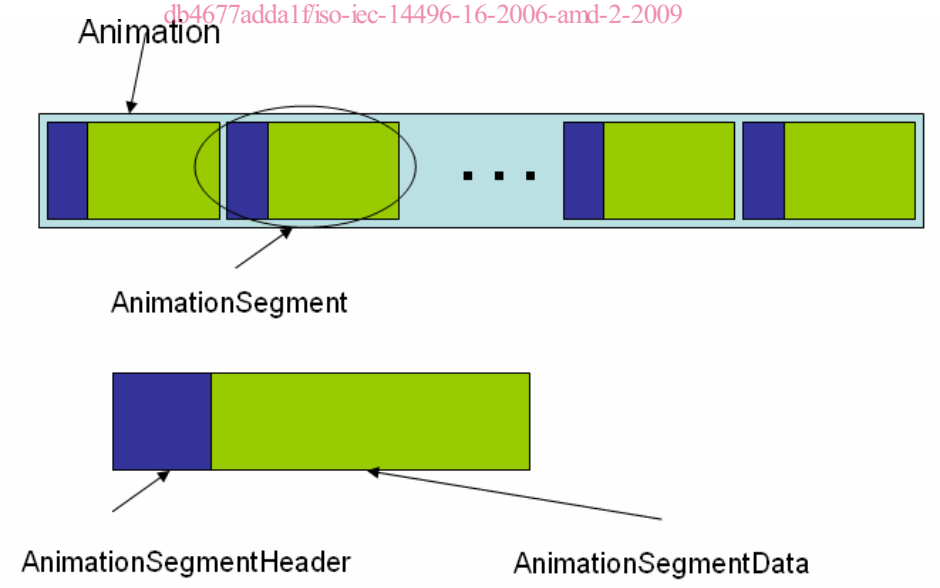

Figure AMD2.1 — FAMC bitstream structure.

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

An animation segment contains two types of information:

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

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

Figure AMD2.2 illustrates the FAMC decoding process.

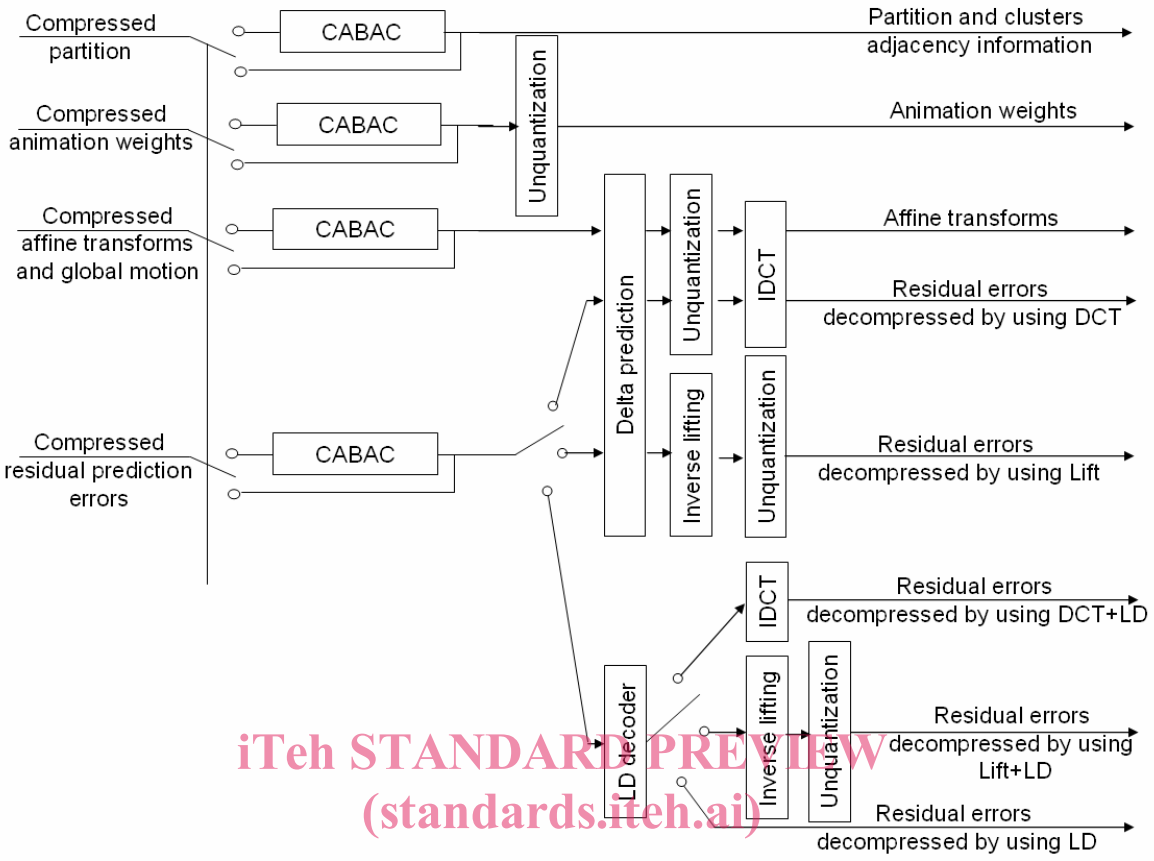

Figure AMD2.2 — FAMC decoding process.

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

5.10.2 FAMC inclusion in the scene graph

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

5.10.3 FAMC class

5.10.3.1 Syntax

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

5.10.3.2 Semantics

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

FAMCAnimationSegmentStartCode = 00 00 01 F0.

5.10.4 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTE

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

5.10.6 FAMCAnimationSegmentData class

5.10.6.1 Syntax

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

5.10.6.2 Semantics

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

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

5.10.7 FAMCSkinningModel class

5.10.7.1 Syntax

```
class FAMCSkinningModel {
   FAMCGlobalTranslationDecoder globalTranslationCompensation;
   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.

5.10.9 FAMCInfoTableDecoder class

5.10.9.1 Syntax

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

5.10.9.2 Semantics

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

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

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

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

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

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

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

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

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

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

5.10.10 FAMCCabacVx3Decoder2

5.10.10.1 Syntax

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

   EGk++;
   cutOff++;
```

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

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

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

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

```
            }
         }
      }
   }

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

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

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

5.10.10.2 Semantics

delta: reciprocal value of the quantization step size.

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

EGk: order of the Exp-Golomb binarization.

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

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

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

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

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

The FAMCCABACDecoder class decodes a (V 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);
}
```