INTERNATIONAL STANDARD

ISO/IEC 14496-16

First edition
2004-02-01

AMENDMENT 1
2006-01-15

Information technology — Coding of audio-visual objects —

Part 16:
Animation Framework eXtension (AFX)

AMENDMENT 1: Morphing and textures

Technologies de l'information — Codage des objets audiovisuels —

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

AMENDEMENT 1: Morphage et textures

Reference number
ISO/IEC 14496-16:2004/Amd.1:2006(E)

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 1 to ISO/IEC 14496-16:2004 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

Information technology — Coding of audio-visual objects —

Part 16: **Animation Framework eXtension (AFX)**

AMENDMENT 1: Morphing and textures

Add subclause 4.3.6 MorphSpace:

4.3.6 MorphSpace

4.3.6.1 Introduction

Morphing is mainly an interpolation technique used to create from two objects a series of intermediate objects that change continuously, in order to make a smooth transition from the source to the target. A straight extension of the morphing between two elements –the source and the target- consists in considering a collection of possible targets and compose a virtual object configuration by weighting those targets. This collection represents a basis of animation space and animation is performed by simply updating the weight vector. The following node allows the representation of a mesh as a combination of a base shape and a collection of target geometries.

4.3.6.2 Node interface

```
MorphShape{ #%NDT=SF3DNode,SF2DNode
      exposedField SFInt32 morphID
      exposedField SFShapeNode baseShape
      exposedField MFShapeNode targetShapes [ ]
      exposedField MFFloat weights [ ]
}
```

4.3.6.3 Semantics

morphID – a unique identifier between 0 and 1023 which allows that the morph to be addressed at animation run-time.

baseShape – a Shape node that represent the base mesh. The geometry field of the baseShape can be any geometry supported by ISOIEC 14496 (e.g. IndexedFaceSet, IndexedLineSet, SolidRep).

targetShapes – a vector of Shapes nodes representing the shape of the target meshes. The tool used for definig an appearance and a geometry of a target shape must be the same as the tool used for defining the appearance and the geometry of the base shape (e.g. if the baseShape is defined by using IndexedFaceSet, all the target shapes must be defined by using IndexedFaceSet).

weights – a vector of integers of the same size as the **targetShapes**. The morphed shape is obtained according to the following formula:

$$M = B + \sum_{i=1}^{n} (T_i - B) * w_i \qquad \text{(ADM1-1)}$$

with

M –morphed shape,

B – base shape,

T_i – target shape *i*,

W_i – weight of the T_i.

The morphing is performed for all the components of the Shape (Appearance and Geometry) that have different values in the base shape and the target shapes [e.g. if the base shape and the target shapes are definined by using IndexedFaceSet and the *coord* field contains different values in the base shape and in the target geometries, the *coord* component of the morph shape is obtained by using Equation (ADM1-1)] applied to the *coord* field. Note that the size of the *coord* field must be the same for the base shapes and the target shapes).

If the shapes (base and targets) are defined by using IndexedFaceSet, a tipical decoder should support morphing of the following geometry components: *coord*, *normals*, *color*, *texCoord*.

Add subclause 4.5.4 Depth Image-based Representation Version 2:

4.5.4 Depth Image-based Representation Version 2

4.5.4.1 Introduction

Version 1 of DIBR introduced depth image-based representations (DIBR) of still and animated 3D objects. Instead of a complex polygonal mesh, which is hard to construct and handle for realistic models, image- or point-based methods represent a 3D object (scene) as a set of reference images completely covering its visible surface. This data is usually accompanied by some kind of information about the object geometry. To that end, each reference image comes with a corresponding depth map, an array of distances from the pixels in the image plane to the object surface. Rendering is achieved by either forward warping or splat rendering. But with Version 1 of the specification of DIBR nodes no high-quality rendering can be achieved.

Version 2 nodes allow for high-quality rendering of depth image-based representations. High-quality rendering is based on the notion of point-sampled surfaces as non-uniformly sampled signals. Point-sampled surfaces can be easily constructed from the DIBR nodes by projecting the pixels with depth into 3D-space. The discrete signals are rendered by reconstructing and band-limiting a continuous signal in image space using so called resampling filters.

A point-based surface consists of a set of non-uniformly distributed samples of a surface; hence we interpret it as a non-uniformly sampled signal. To continuously reconstruct this signal, we have to associate a 2D reconstruction kernel $r_k(u)$ with each sample point p_k . The kernels are defined in a local tangent frame with coordinates $u = (u, v)$ at the point p_k , as illustrated on the left in Figure AMD1-1. The tangent frame is defined by the splat and normal extensions of the DIBR structures Version 2 [1].

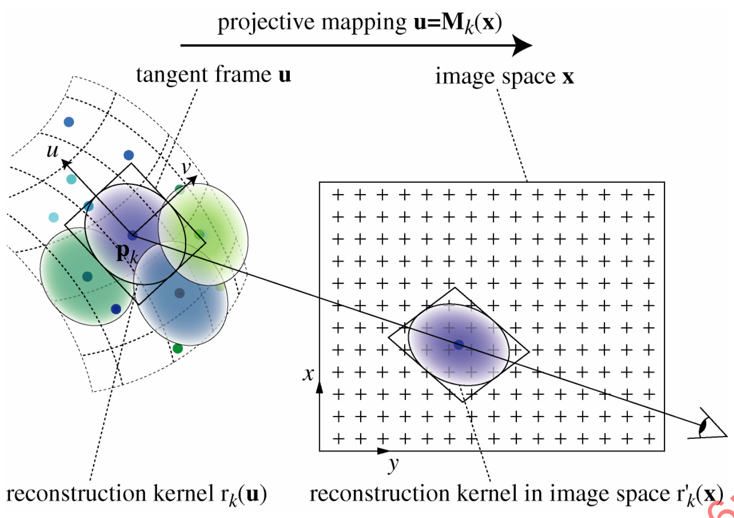
projective mapping $\mathbf{u}=\mathbf{M}_k(\mathbf{x})$
tangent frame $\mathbf{u}$
image space $\mathbf{x}$
u
v
$\mathbf{p}_k$
x
y
reconstruction kernel $r_k(\mathbf{u})$
reconstruction kernel in image space $r'_k(\mathbf{x})$

Figure AMD1-1 — Local tangent planes and reconstruction kernels

4.5.4.2 DepthImageV2 Node

4.5.4.2.1 Node interface

```
DepthImageV2 { #%NDT=SF3DNode
        exposedField       SFVec3f              position       0 0 10
        exposedField       SFRotation           orientation    0 0 1 0
        exposedField       SFVec2f              fieldOfView    π/4 π/4
        exposedField       SFFloat              nearPlane      10
        exposedField       SFFloat              farPlane       100
        field              SFVec2f              splatMinMax    0.1115 0.9875
        exposedField       SFBool               orthographic   TRUE
        field              SFDepthTextureNode   diTexture      NULL
}
```

4.5.4.2.2 Functionality and semantics

The **DepthImageV2** node defines a single IBR texture. When multiple **DepthImage** nodes are related to each other, they are processed as a group, and thus, should be placed under the same Transform node.

The **diTexture** field specifies the texture with depth, which shall be mapped into the region defined in the **DepthImageV2** node. It shall be one of the various types of depth image texture (**SimpleTextureV2** or **PointTextureV2**).

The **position** and **orientation** fields specify the relative location of the viewpoint of the IBR texture in the local coordinate system. **position** is relative to the coordinate system's origin (0, 0, 0), while **orientation** specifies a rotation relative to the default orientation. In the default position and orientation, the viewer is on the Z-axis looking down the –Z-axis toward the origin with +X to the right and +Y straight up. However, the transformation hierarchy affects the final position and orientation of the viewpoint.

The **fieldOfView** field specifies a viewing angle from the camera viewpoint defined by **position** and **orientation** fields. The first value denotes the angle to the horizontal side and the second value denotes the angle to the vertical side. The default values are 45 degrees in radians. However, when **orthographic** field is set to TRUE, the **fieldOfView** field denotes the width and height of the near plane and far plane.

The **nearPlane** and **farPlane** fields specify the distances from the viewpoint to the near plane and far plane of the visibility area. The texture and depth data shows the area closed by the near plane, far plane and the **fieldOfView**. The depth data are scaled to the distance from **nearPlane** to **farPlane**.

The **splatMinMax** field specifies the minimum and maximum splat vector lengths. The splatU and splatV data of SimpleTextureV2 is scaled to the interval defined by the splatMinMax field.

The **orthographic** field specifies the view type of the IBR texture. When set to TRUE, the IBR texture is based on orthographic view. Otherwise, the IBR texture is based on perspective view.

The **position, orientation, fieldOfView, nearPlane, farPlane,** and **orthographic** fields are exposedField types, which are for extrinsic parameters. The DepthImage node supports the camera movement and changeable view frustum corresponding to movement or deformation of a DIBR object.

Reference images that are suitable to the characteristic of a DIBR model are obtained in the modeling stage. Therefore, the fields that reflect the camera movement and the changeable view frustum and the reference images in the modeling stage are used to create a view frustum and a DIBR object in the rendering stage.

4.5.4.3 SimpleTextureV2 node

4.5.4.3.1 Node interface

```
SimpleTextureV2 { #%NDT=SFDepthTextureNode
        field        SFTextureNode        texture        NULL
        field        SFTextureNode        depth          NULL
        field        SFTextureNode        normal         NULL
        field        SFTextureNode        splatU         NULL
        field        SFTextureNode        splatV         NULL
}
```

4.5.4.3.2 Functionality and semantics

The **SimpleTextureV2** node defines a single layer of IBR texture.

The **texture** field specifies the flat image that contains color for each pixel. It shall be one of the various types of texture nodes (**ImageTexture**, **MovieTexture** or **PixelTexture**).

The **depth** field specifies the depth for each pixel in the **texture** field. The size of the depth map shall be the same size as the image or movie in the **texture** field. Depth field shall be one of the various types of texture nodes (ImageTexture, MovieTexture or PixelTexture), where only the nodes representing gray scale images are allowed. If the depth field is unspecified, the alpha channel in the texture field shall be used as the depth map. If the depth map is not specified through depth field or alpha channel, the result is undefined.

Depth field allows to compute the actual distance of the 3D points of the model to the plane which passes through the viewpoint and parallel to the near plane and far plane:

$$dist = nearPlane + \left(1 - \frac{d-1}{d_{\max}-1}\right)(farPlane - nearPlane). \quad \text{(AMD1-2)}$$

where *d* is depth value and d_{max} is maximum allowed depth value. It is assumed that for the points of the model, $d>0$, where $d=1$ corresponds to far plane, $d=d_{max}$ corresponds to near plane.

This formula is valid for both perspective and orthographic case, since *d* is distance between the point and the plane. max *d* is the largest *d* value that can be represented by the bits used for each pixel:

(1) If the depth is specified through **depth** field, then depth value *d* equals to the gray scale.

(2) If the depth is specified through alpha channel in the image defined via **texture** field, then the depth value *d* is equal to alpha channel value.

The depth value is also used to indicate which points belong to the model: only the point for which d is nonzero belong to the model.

For animated DepthImage-based model, only DepthImage with SimpleTextures as diTextures are used.

Each of the Simple Textures can be animated in one of the following ways:

(1) **depth** field is still image satisfying the above condition, **texture** field is arbitrary MovieTexture

(2) **depth** field is arbitrary MovieTexture satisfying the above condition on the **depth** field, **texture** field is still image

(3) both **depth** and **texture** are MovieTextures, and **depth** field satisfies the above condition

(4) **depth** field is not used, and the depth information is retrieved from the alpha channel of the MovieTexture that animates the **texture** field

The **normal** field specifies the normal vector for each pixel in the **texture** field. The normal vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. The normal map shall be the same size as the image or movie in the **texture** field. Normal field shall be one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture), where only the nodes representing color images are allowed. If the normal map is not specified through the normal field, the decoder can calculate a normal field by evaluating the cross-product of the splatU and splatV fields. If neither the normal map nor the splatU and splatV fields are specified, only basic rendering is possible.

The **splatU** and **splatV** fields specify the tangent plane and reconstruction kernel needed for high-quality point-based rendering. Both splatU and splatV fields have to be scaled to the interval defined by the splatMinMax field.

The **splatU** field specifies the splatU vector for each pixel in the **texture** field. The splatU vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. The splatU map shall be the same size as the image or movie in the **texture** field. splatU field shall be one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture), where the nodes either representing color or gray scale images are allowed. If the splatU map is specified as gray scale image the decoder can calculate a circular splat by using the normal map to produce a tangent plane and the splatU map as radius. In this case, if the normal map is not specified, the result is undefined. If the splatU map is specified as color image, it can be used in conjunction with the splatV map to calculate a tangent frame and reconstruction kernel for high-quality point-based rendering. If neither the normal map nor the splatV map is specified, the result is undefined.

The **splatV** field specifies the splatV vector for each pixel in the **texture** field. The splatV vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. The splatV map shall be the same size as the image or movie in the **texture** field. splatV field shall be one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture), where only the nodes representing color images are allowed. If the splatU map is not specified as well, the result is undefined.

4.5.4.4 PointTextureV2 node

4.5.4.4.1 Node interface

```
PointTextureV2 { #%NDT=SFDepthTextureNode
      field       SFInt32           width            256
      field       SFInt32           height           256
      field       SFInt32           depthNbBits      7
      field       MFInt32           depth            []
      field       MFColor           color            []
      field       SFNormalNode      normal
      field       MFVec3f           splatU           []
      field       MFVec3f           splatV           []
}
```

4.5.4.4.2 Functionality and semantics

The **PointTextureV2** node defines multiple layers of IBR points.

The **width** and **height** field specify the width and height of the texture.

Geometrical meaning of the depth values, and all the conventions on their interpretation adopted for the SimpleTexture, apply here as well.

The **depth** field specifies a multiple depths of each point in the projection plane, which is assumed to be farPlane (see above) in the order of traversal, which starts from the point in the lower left corner and traverses to the right to finish the horizontal line before moving to the upper line. For each point, the number of depths (pixels) is first stored and that number of depth values shall follow.

The **color** field specifies color of current pixel. The order shall be the same as the **depth** field except that number of depths (pixels) for each point is not included.

The **depthNbBits** field specifies the number of bits used for the original depth data. The value of depthNbBits ranges from 0 to 31, and the actual number of bits used in the original data is depthNbBits+1. The d_{max} used in the distance equation is derived as follows:

$$d_{\max} = 2^{(depthNbBits+1)} - 1. \quad \text{(AMD1-3)}$$

The **normal** field specifies normals for each specified depth of each point in the projection plane in the same order. The normal vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. If the normals are not specified through the normal field, the decoder can calculate a normal field by evaluating the cross-product of the splatU and splatV fields. If neither the normals nor the splatU and splatV fields are specified, only basic point rendering is possible. Normals can be quantized by using the SFNormalNode functionality.

The **splatU** field specifies splatU vectors for each specified depth of each point in the projection plane in the same order. The splatU vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. If the splatV vectors are not specified the decoder can calculate a circular splat by using the normals to produce a tangent plane and the length of the splatU vectors as radius. In this case, if the normals are not specified, the result is undefined. If the splatU vectors are specified, it can be used in conjunction with the splatV vectors to calculate a tangent frame and reconstruction kernel for high-quality point-based rendering. If neither the normals nor the splatV vectors are specified, the result is undefined.

The **splatV** field specifies splatV vectors for each specified depth of each point in the projection plane in the same order. The splatV vector should be assigned to the object-space point sample derived from extruding the pixel with depth to 3-space. If the splatU vectors are not specified as well, the result is undefined.

Add subclause 4.5.5 Multitexturing:

4.5.5 Multitexturing

4.5.5.1 MultiTexture Node

4.5.5.1.1 Node interface

```
MultiTexture { #%NDT=SFTextureNode
  exposedField SFFloat          alpha        1     #[0,1]
  exposedField SFColor          color        1 1 1     #[0,1]
  exposedField MFInt        function         []
  exposedField MFInt        mode             []
  exposedField MFInt        source           []
  exposedField MFTextureNode      texture       []
  exposedField MFVec3f        cameraVector    []
  exposedField SFBool           transparent     FALSE
}
```

4.5.5.1.2 Functionality and semantics

MultiTexture enables the application of several individual textures to a 3D object to achieve a more complex visual effect. MultiTexture can be used as a value for the texture field in an Appearance node.

The **texture** field contains a list of texture nodes (e.g., ImageTexture, PixelTexture, MovieTexture). The texture field may not contain another MultiTexture node.

The **cameraVector** field contains a list of camera vectors in 3D for each texture in the **texture** field. These vectors point from each associated camera to the 3D scene center. The view vectors are used to calculate texture weights according to the unstructured lumigraph approach from [1] for each render cycle, to weight all textures according to the actual scene viewpoint.

The **color** and **alpha** fields define base RGB and alpha values for SELECT mode operations.

The **mode** field controls the type of blending operation. The available modes include MODULATE for a lit Appearance, REPLACE for an unlit Appearance and several variations of the two. However, for view-dependent Multitexturing the default mode MODULATE shall be used in conjuction with the **source** field value "FACTOR".

Table AMD1-1 lists possible multitexture modes.

Table AMD1-1 — Multitexture modes

VALUE	MODE	Description
00000	MODULATE	Multiply texture color with current color Arg1 × Arg2
00001	REPLACE	Replace current color Arg2
00010	MODULATE2X	Multiply the components of the arguments, and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening.
00011	MODULATE4X	Multiply the components of the arguments, and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening.
00100	ADD	Add the components of the arguments Arg1 + Arg2
00101	ADDSIGNED	Add the components of the arguments with a -0.5 bias, making the effective range of values from −0.5 through 0.5.
00110	ADDSIGNED2X	Add the components of the arguments with a -0.5 bias, and shift the products to the left 1 bit.
00111	SUBTRACT	Subtract the components of the second argument from those of the first argument. Arg1 − Arg2
01000	ADDSMOOTH	Add the first and second arguments, then subtract their product from the sum. Arg1 + Arg2 − Arg1 × Arg2 = Arg1 + (1 − Arg1) × Arg2
01001	BLENDDIFFUSEALPHA	Linearly blend this texture stage, using the interpolated alpha from each vertex. Arg1 × (Alpha) + Arg2 × (1 − Alpha)

VALUE	MODE	Description
01010	BLENDTEXTUREALPHA	Linearly blend this texture stage, using the alpha from this stage's texture. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01011	BLENDFACTORALPHA	Linearly blend this texture stage, using the alpha factor from the MultiTexture node. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01100	BLENDCURRENTALPHA	Linearly blend this texture stage, using the alpha taken from the previous texture stage. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01101	MODULATEALPHA_ADDCOLOR	Modulate the color of the second argument, using the alpha of the first argument; then add the result to argument one. Arg1.RGB + Arg1.A × Arg2.RGB
01110	MODULATEINVALPHA_ADDCOLOR	Similar to MODULATEALPHA_ADDCOLOR, but use the inverse of the alpha of the first argument. (1 − Arg1.A) × Arg2.RGB + Arg1.RGB
01111	MODULATEINVCOLOR_ADDALPHA	Similar to MODULATECOLOR_ADDALPHA, but use the inverse of the color of the first argument. (1 − Arg1.RGB) × Arg2.RGB + Arg1.A
10000	OFF	Turn off the texture unit
10001	SELECTARG1	Use color argument 1 Arg1
10010	SELECTARG2	Use color argument 1 Arg2
10011	DOTPRODUCT3	Modulate the components of each argument (as signed components), add their products, then replicate the sum to all color channels, including alpha. This can do either diffuse or specular bump mapping with correct input. Performs the function (Arg1.R × Arg2.R + Arg1.G × Arg2.G + Arg1.B × Arg2.B) where each component has been scaled and offset to make it signed. The result is replicated into all four (including alpha) channels.
10100 – 11111		Reserved for future use

The **source** field determines the color source for the second argument. Table AMD1-2 lists valid values for the source field. For view-dependent Multitexturing “FACTOR” shall be used in conjuction with the **mode** field value MODULATE.

Table AMD1-2 — Values for the *source* field

VALUE	MODE	Description
000	"" (default)	The second argument color (ARG2) is the color from the previous rendering stage (DIFFUSE for first stage).
001	"DIFFUSE"	The texture argument is the diffuse color interpolated from vertex components during Gouraud shading.
010	"SPECULAR"	The texture argument is the specular color interpolated from vertex components during Gouraud shading.
011	"FACTOR"	The texture argument is the factor (color, alpha) from the MultiTexture node.
100-111		Reserved for future use

The **function** field defines an optional function to be applied to the argument after the mode has been evaluated. Table AMD1-3 lists valid values for the **function** field.

Table AMD1-3 — Values for the *function* field

VALUE	MODE	Description
000	"" (default)	No function is applied.
001	"COMPLEMENT"	Invert the argument so that, if the result of the argument were referred to by the variable x, the value would be 1.0 minus x.
010	"ALPHAREPLICATE"	Replicate the alpha information to all color channels before the operation completes.
011-111		Reserved for future use

Mode may contain an additional Blending mode for the alpha channel; e.g., "MODULATE,REPLACE" specifies Color = (Arg1.color × Arg2.color, Arg1.alpha).

The number of used texture stages is determined by the length of the texture field. If there are fewer mode values, the default mode is "MODULATE".

Note: Due to the texture stage architecture and its processing order of textures in common graphic cards, the result of general texture weighting depends on the order of textures if more than two textures are used. If order-independent texture mapping is required, the proposed settings can be used, i.e. MODULATE for the **mode** field and “TFACTOR” for the **source** field.

Beside the **MultiTexture**-Node that assigns the actual 2D images to the scene, contains blending modes and transform parameters, the second component of Multi-Texturing is the **MultiTextureCoordinate**-Node. This node addresses the relative 2D coordinates of each texture, which are combined with the 3D points of the scene geometry. In Multi-Texturing, one 3D point is associated with *n* 2D texture points with *n* being the number of views. The node syntax for **MultiTextureCoordinate** in X3D is as follows and can be used as is.

4.5.5.2 MultiTextureCoordinate Node

MultiTextureCoordinate node supplies multiple texture coordinates per vertex. This node can be used to set the texture coordinates for the different texture channels.

4.5.5.2.1 Node interface

```
MultiTextureCoordinate { #%NDT=SFTextureCoordinateNode
  exposedField MFTextureCoordinateNode texCoord []
}
```

4.5.5.2.2 Functionality and semantics

Each entry in the **texCoord** field may contain a **TextureCoordinate** or **TextureCoordinateGenerator** node.

By default, if using **MultiTexture** with an **IndexedFaceSet** without a **MultiTextureCoordinate texCoord** node, texture coordinates for channel 0 are replicated along the other channels. Likewise, if there are too few entries in the **texCoord** field, the last entry is replicated.

Example:

```
Shape {
  appearance Appearance {
    texture MultiTexture {
      mode [ 0 0 0 0 ]
      source [ 3 3 3 3 ]
      texture [
        ImageTexture { url "np00.jpg" }
        ImageTexture { url "np01.jpg" }
        ImageTexture { url "np02.jpg" }
        ImageTexture { url "np03.jpg" }
```

```
          ]
        }
      }
      geometry IndexedFaceSet {
        ...
        texCoord MultiTextureCoord {
          texCoord [
            TextureCoordinate { ... }
            TextureCoordinate { ... }
            TextureCoordinate { ... }
            TextureCoordinate { ... }
          ]
        }
      }
    }
```

Add subclause 4.7.1.7, SBVCAnimationV2:

4.7.1.7.1 Introduction

This node is an extension of the SBVCAnimation node and the added functionality consists in streaming control and animation data collection. The BBA stream can be controlled as a elementary media stream, and can be used in connection with the MediaControl node.

4.7.1.7.2 Syntax

```
SBVCAnimationV2{ #%NDT=SF3DNode,SF2DNode
      exposedField    MFNode   virtualCharacters  [ ]
      exposedField    MFURL    url                [ ]
      exposedField    SFBool   loop               FALSE
      exposedField    SFFloat  speed              1.0
      exposedField    SFTime   startTime          0
      exposedField    SFTime   stopTime           0
      eventOut        SFTime   duration_changed
      eventOut        SFBool   isActive
      exposedField    MFInt    activeUrlIndex     []
      exposedField    SFFloat  transitionTime     0
}
```

4.7.1.7.3 Semantics

The **virtualCharacters** field specifies a list of SBSkinnedModel nodes. The length of the list can be 1 or greater.

The **url** field refers to the BBA stream which contains encoded animation data related to the SBSkinnedModel nodes from the virtualCharacters list and is used for outband bitstreams. The animation will be extracted from the first element of the animation URL list and if the case when it is not available the following element will be used.

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the SBVCAnimationV2 node, are similar with the ones described by VRML specifications (ISO/IEC 14772-1:1997) for AudioClip, MovieTexture, and TimeSensor nodes and are described as follows.

The values of the exposedFields are used to determine when the node becomes active or inactive.

The SBVCAnimationV2 node can execute for 0 or more cycles. A cycle is defined by field data within the node. If, at the end of a cycle, the value of **loop** is FALSE, execution is terminated. Conversely, if **loop** is TRUE at the end of a cycle, a time-dependent node continues execution into the next cycle. A time-dependent node with **loop** TRUE at the end of every cycle continues cycling forever if **startTime** >= **stopTime**, or until **stopTime** if **startTime** < **stopTime**.

The SBVCAnimationV2 node generates an **isActive** TRUE event when it becomes active and generates an **isActive** FALSE event when it becomes inactive. These are the only times at which an **isActive** event is generated. In particular, **isActive** events are not sent at each tick of a simulation.

The SBVCAnimationV2 node is inactive until its **startTime** is reached. When time *now* becomes greater than or equal to **startTime**, an **isActive** TRUE event is generated and the SBVCAnimationV2 node becomes active (*now* refers to the time at which the player is simulating and displaying the virtual world). When a SBVCAnimationV2 node is read from a mp4 file and the ROUTEs specified within the mp4 file have been established, the node should determine if it is active and, if so, generate an **isActive** TRUE event and begin generating any other necessary events. However, if a SBVCAnimationV2 node would have become inactive at any time before the reading of the mp4 file, no events are generated upon the completion of the read.

An active SBVCAnimationV2 node will become inactive when **stopTime** is reached if **stopTime** > **startTime**. The value of **stopTime** is ignored if **stopTime** <= **startTime**. Also, an active SBVCAnimationV2 node will become inactive at the end of the current cycle if **loop** is FALSE. If an active SBVCAnimationV2 node receives a *set_loop* FALSE event, execution continues until the end of the current cycle or until **stopTime** (if **stopTime** > **startTime**), whichever occurs first. The termination at the end of cycle can be overridden by a subsequent *set_loop* TRUE event.

Any *set_startTime* events to an active SBVCAnimationV2 node are ignored. Any *set_stopTime* event where **stopTime** <= **startTime** sent to an active SBVCAnimationV2 node is also ignored. A *set_stopTime* event where **startTime** < **stopTime** <= *now* sent to an active SBVCAnimationV2 node results in events being generated as if **stopTime** has just been reached. That is, final events, including an **isActive** FALSE, are generated and the node becomes inactive. The **stopTime**_*changed* event will have the *set_stopTime* value.

A SBVCAnimationV2 node may be restarted while it is active by sending a *set_stopTime* event equal to the current time (which will cause the node to become inactive) and a *set_startTime* event, setting it to the current time or any time in the future. These events will have the same time stamp and should be processed as *set_stopTime,* then *set_startTime* to produce the correct behaviour.

The **speed** exposedField controls playback speed. It does not affect the delivery of the stream attached to the **SBVCAnimationV2** node. For streaming media, value of **speed** other than 1 shall be ignored.

A **SBVCAnimationV2** shall display first frame if **speed** is 0. For positive values of **speed**, the frame that an active **SBVCAnimationV2** will display at time *now* corresponds to the frame at animation time (i.e., in the animation's local time base with frame 0 at time 0, at speed = 1):

fmod (now - **startTime**, duration/**speed**)

If **speed** is negative, then the frame to display is the frame at animation time:

duration + fmod(now - **startTime**, duration/**speed**).

When a **SBVCAnimationV2** becomes inactive, the frame corresponding to the time at which the **SBVCAnimationV2** became inactive shall persist. The **speed** exposedField indicates how fast the movie shall be played. A speed of 2 indicates the animation plays twice as fast. Note that the **duration_changed** eventOut is not affected by the **speed** exposedField. **set_speed** events shall be ignored while the animation is playing.

An event shall be generated via the **duration_changed** field whenever a change is made to the **startTime** or **stopTime** fields. An event shall also be triggered if these fields are changed simultaneously, even if the duration does not actually change.

activeUrlIndex allows to select or to combine specific animation resource referred in the **url[]** field. When this field is instantiated the behavior of the **url[]** field changes from the alternative selection into a combined selection. In the case of alternative mode, if the first resource in the **url[]** field is not available, the second one will be used, and so on. In the combined mode the following cases can occur:

(1) **activeUrlIndex** has one field: the resource from **url[]** that has this index is used for animation. When the **activeUrlIndex** is updated a transition between to the old animation (frame) and the new one is performed. The transition use linear interpolation for translation, center and scale and SLERP for spherical data as rotation and scaleOrientation. The time of transition is specified by using the **transitionTime** field.

(2) **activeUrlIndex** has several fields: a composition between the two resources is performed by the terminal: for the bones that are common in two or more resources a mean procedure has to be applied. The mean is computed by using linear interpolation for translation, center and scale and SLERP for spherical data as rotation and scaleOrientation.

In all the cases, when a transition between two animation resources is needed, when the **transitionTime** is not zero, a interpolation must be performed by the player. The **transitionTime** is specified in miliseconds.

In subclause 5.4.2.1 replace:

The BBA stream is an extended form of the FBA stream [2]. See ISO/IEC 14496-2, Coding of Audio-Visual Objects:Visual [2] for details on FBA stream. A BBA Object plane contains the update values for Skin&Bones components (SBC) which can be translations in one direction, rotation angles, scale factors in an arbitrary direction (for bones) and control points translation, control points weights factors or (and) knots (for muscles).

with:

The BBA stream is an extended form of the FBA stream [2]. See ISO/IEC 14496-2, Coding of Audio-Visual Objects:Visual [2] for details on FBA stream. A BBA Object plane contains the update values for Skin&Bones components (SBC) which can be translations, rotation angles, scale factors in an arbitrary direction - for bones, control points translation, control points weights factors, knots - for muscles, and weights of the target meshes – for morphing.

In subclause 5.4.2.3.2 replace:

The bba_object_plane is the access unit of the BBA stream. It contains the bba_object_plane_header, which specifies timing, and the bba_object_plane_data, which contains the data for all nodes (bones & muscles) being animated.

with:

The bba_object_plane is the access unit of the BBA stream. It contains the bba_object_plane_header, which specifies timing, and the bba_object_plane_data, which contains the data for all nodes (bones, muscles and morphs) being animated.

Replace subclause 5.4.2.8.1 with:

```
class bba_object_plane_mask() {
        bit(5) NumberOfInterpolatedFrames; //NIF
        if (isIntra){
         bit(5) bba_quant;
         bit(3) pow2quant;
         bit(3) noOfControllerTypes;
         bit(10) NumberOfBones; //NSBB
         bit(10) NumberOfMuscles;//NSBM
         bit(10) NumberOfMorphs;//NMF
         for (bone=1;bone<NumberOfBones;bone++){
                 bit(10) BoneIdentifier; //IDB
                 bone_mask bnmask();
         }
         for (ms=1;ms< NumberOfMuscles;ms+){
                 bit(10) MuscleIdentifier; //IDM
```

```
                        bit(6) NumberControlPoints; //NCP
                        bit(6) NumberKnots; //NK
                        muscle_mask msmask();
                }
                for (mf=1;mf< NumberOfMorphs;mf+){
                        bit(10) MorphIdentifier; //IDMF
                        bit(6) NumberOfWeights; //NW
                        morph_mask mfmask();
                }
        }
    }
```

In subclause 5.4.2.8.2, add:

noOfControllerTypes – a 3 bits integer indicating the number of controller types in this version of the BBA stream (here this value must be “3”)

NumberOfMorphs – a 10 bits integer indicating the number of morph objects that are animated in the current frame

MorphIdentifier – a 10 bits unique identifier that indicate what morph is currently animated; this value must be identical with the MorphID field from the MorphShape Node

NumberOfWeights – a 6 bits integer indicating the number the weights of the current morph objects

Add subclause 5.4.2.11 morph_mask:

5.4.2.11 morph_mask

5.4.2.11.1 Syntax

```
class morph_mask() {
        for (w=1;w<=NW;w++){
                bit(1) marker_bit;
                bit(1) IsWeight_changed[w];
        }
}
```

5.4.2.11.2 Semantics

IsWeight_changed – a vector with the same dimention as the weights field from the MorphShape node indicating if the weight of a target shape is changed in the current animation frame.

Change subclause 5.4.2.11 bba_object_plane_values with 5.4.2.12 bba_object_plane_values.

In subclause 5.4.2.12.2 Semantics, replace:

NUM_SBCs=NSBB*16+NSBM*(NCP*3+NCP+NK)

with:

$$NUM\_SBCs = NSBB * 16 + \sum_{ms=1}^{NSBM} (3 * NCP_{ms} + NCP_{ms} + NK_{ms}) + \sum_{mf=1}^{NMF} NW_{mf} \qquad \textbf{(ADM1-4)}$$

Change subclause 5.4.2.12 bba_new_minmax with 5.4.2.13 bba_new_minmax.

Change subclause 5.4.2.13 bba_i_frame with 5.4.2.14 bba_i_frame.

Change subclause 5.4.2.14 bba_p_frame with 5.4.2.15 bba_p_frame.

Change subclause 5.4.2.15 bba_i_segment with 5.4.2.16 bba_i_segment.

In subclause 5.4.2.16.2 Semantics, replace:

See subclause 5.4.2.15

with:

See subclause 5.4.2.16

Change subclause 5.4.2.16 bba_p_segment with 5.4.2.17 bba_p_segment.

In subclause 5.4.2.17.2 Semantics, replace:

See subclause 5.4.2.15

with:

See subclause 5.4.2.16

Change subclause 5.4.2.17 decode_i_dc with 5.4.2.18 decode_i_dc.

In subclause 5.4.2.18.2 Semantics, replace:

See subclause 5.4.2.15

with:

See subclause 5.4.2.16

Change subclause 5.4.2.18 decode_p_dc with 5.4.2.19 decode_p_dc.

In subclause 5.4.2.19.2 Semantics replace:

See subclause 5.4.2.15

with:

See subclause 5.4.2.16

Change subclause 5.4.2.19 decode_ac with 5.4.2.20 decode_ac.

Add subclause 5.3.2 PointTexture Compression Specification:

5.3.2 PointTexture Compression

5.3.2.1 Overview

The PointTexture compression is a tool to compress the PointTexture node efficiently. The decoder structure of the PointTexture compression is shown in **Figure AMD1-2**. The PointTexture decoder consists of header decoder and node decoder. The header information is decoded in the header decoder and is used in node decoder. The PointTexture decoder receives the arithmetic coded bitstream and restores the PointTexture node. The decoded PointTexture node in ISO/IEC 14496-16 has the depth information and the color information.

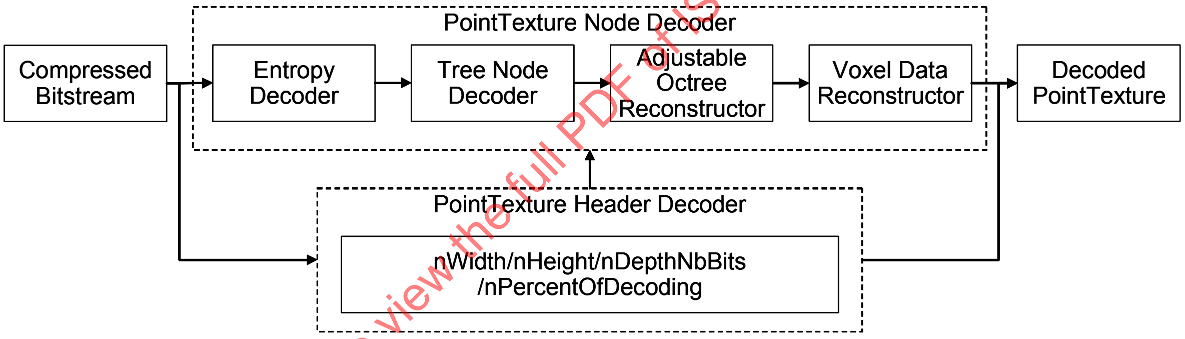
PointTexture Node Decoder
Compressed Bitstream
Entropy Decoder
Tree Node Decoder
Adjustable Octree Reconstructor
Voxel Data Reconstructor
Decoded PointTexture
PointTexture Header Decoder
nWidth/nHeight/nDepthNbBits /nPercentOfDecoding

Figure AMD1-2 — Block diagram of decoder for PointTexture compression

5.3.2.2 PointTexture class

5.3.2.2.1 Syntax

```
class PointTexture ()
{
    PointTextureHeader ();
    PointTextureTreeNodes ();
}
```

5.3.2.2.2 Semantics

This is a top class for reading the compressed bitstream of PointTexture. PointTextureHeader is the class for reading header information from the bitstream. PointTextureTreeNodes is the class for reading tree node information progressively from low to high resolution.

5.3.2.3 PointTextureHeader class

5.3.2.3.1 Syntax

```
class PointTextureHeader ()
{
    unsigned int(5) nBitSizeOfWidth;
    unsigned int(nBitSizeOfWidth) nWidth;
    unsigned int(5) nBitSizeOfHeight;
    unsigned int(nBitSizeOfHeight) nHeight;
    unsigned int(5) nDepthNbBits;
    unsigned int(7) nPercentOfDecoding;
}
```

5.3.2.3.2 Semantics

nBitSizeOfWidth: This value indicates the bit size of nWidth.

nWidth: This value indicates the width of the PointTexture.

nBitSizeOfHeight: This value indicates the bit size of nHeight.

nHeight: This value indicates the height of the PointTexture.

nDepthNbBits: This value indicates the number of bits used for representing the original depth data. The value of nDepthNbBits ranges from 0 to 31, and the number of bits used in the original data is nDepthNbBits + 1.

nPercentOfDecoding: This value indicates the percent of the tree nodes to be decoded. If the value is the maximum (100), the lossless decoding is performed. Otherwise, the lossy decoding is performed.

5.3.2.4 PointTextureTreeNodes class

5.3.2.4.1 Syntax

```
class PointTextureTreeNodes ()
{
    nNumberOfTreeNodes = initializeOctree(nWidth, nHeight, nDepthNbBits);
    nNumberLimit = nNumberOfTreeNodes * nPercentOfDecoding / 100;
    pushQ(0);// 0: root
    nCount = 0;

    while(nCount < nNumberLimit)
    {
      if(isQueueEmpty() == true)      // break if queue is empty
        break;

      nIndex = popQ();
      nCount++;

      nSOP = decodeAAC(contextSOP);
      if(nSOP == 0)    // Split node decoding
      {
        nRegionRed = decodeAAC(contextRedOfRegion);
        nRegionGreen = decodeAAC(contextGreenOfRegion);
        nRegionBlue = decodeAAC(contextBlueOfRegion);
        for(nChild = 1; nChild <= 8; nChild++)    // 8 children nodes
        {
                nBOW = decodeAAC(contextBOW);   // black or white
                if(nBOW == 0)      // 0: white node
                  nCount += getCountOfTreeSize(nIndex*8+nChild);
                else        // 1: black node
                  pushQ(nIndex*8+nChild);
              }
            }
            else   // PPM node decoding
            {
              getRegion(nIndex, nStartX, nStartY, nStartZ, nEndX, nEndY, nEndZ);
              for(k = nStartZ; k < nEndZ; k++)
              {
                for(j = nStartY; j < nEndY; j++)
                {
                  for(i = nStartX; i < nEndX; i++)
                  {
                    nIndexOfContext = getIndexOfContext(i, j, k);
                    nVoxel = decodeAAC(contextTreeNodes[nIndexOfContext]);
                    if(nVoxel == 1)       // 1: black node
                    {
                      nDeltaRed = decodeAAC(contextColorDifference);
                     nDeltaGreen = decodeAAC(contextColorDifference);
                     nDeltaBlue = decodeAAC(contextColorDifference);
                    }
                  }
                }
              }
              nCount += getCountOfTreeSize(nIndex) - 1;
            }
          }
        }
```

5.3.2.4.2 Semantics

nNumberOfTreeNodes: This value indicates the number of the tree nodes in an octree.

initializeOctree: This function initialize the resolution values with nWidth, nHeight and nDepthNbBits and gets the number of the tree nodes in the octree.

nNumberLimit: This value indicates the limit of the tree nodes to be decoded.

pushQ: This function inserts a value into a queue.

nCount: This value indicates the current number of decoding tree node.

isQueueEmpty: This function checks whether the queue is empty or not.

nIndex: This value indicates the index of the tree node to be decoded.

popQ: This function extracts a value from the queue.

nSOP: This value indicates whether the tree node is split node or PPM(Prediction by Partial Matching) node. If the value is 0, it means split node. Otherwise, it means ppm node.

decodeAAC: This function performs the AAC(Adaptive Arithmetic Coder) decoding with a given context.

nRegionRed: This value indicates the red color range in a voxel region.

nRegionGreen: This value indicates the green color range in a voxel region.

nRegionBlue: This value indicates the blue color range in a voxel region.

nChild: This value indicates the index of the 8 children nodes decoding the split node.

nBOW: This value indicates whether the child node is black or white.

getCountOfTreeSize: This function calculates the number of sub-tree nodes from a tree node.

getRegion: This function calculates the volume region (starting x, y, z and ending x, y, z) from an index of the tree node.

nStartX, nStartY, nStartZ: These values indicate the starting points of the volume region.

nEndX, nEndY, nEndZ: These values indicate the ending points of the volume region.

nIndexOfContext: This value indicates an index of the tree node context from x, y, z values.

getIndexOfContext: This function gets the index of the tree node context from x, y, z values.

nVoxel: This value indicates whether the voxel node is black or white.

nDeltaRed: This value indicates the differentiated value of the red color in a voxel.

nDeltaGreen: This value indicates the differentiated value of the green color in a voxel.

nDeltaBlue: This value indicates the differentiated value of the blue color in a voxel.

5.3.2.5 Decoding Process

5.3.2.5.1 Overview

As shown in Figure AMD1-2, there are two parts to decode PointTexture. Those are header decoder and node decoder. The header decoder is to get the resolution information of PointTexture and the percent value how many tree nodes to decode. And the node decoding process is comprised of the following steps:

- Entropy decoding
- Tree node decoding
- Adjustable octree reconstruction
- Voxel data reconstruction

5.3.2.5.2 Header Decoding

From the nDepthNbBits, the real range of the depth can be obtained as follows.

$$nDepth = 2^{nDepthNbBits + 1}.$$

The resolution of PointTexture is nWidth × nHeight × nDepth. From the resolution values, the adjustable octree can be obtained. The adjustable octree has the five labels as follows:

Table AMD1-4 — Five labels for adjustable octree nodes

Labels	Comments
S	**Split:** The node is subdivided into 8 nodes
W	**White:** The node consists of all white voxels
B	**Fill black:** The node consists of, or is approximated by, all black voxels
P	**PPM:** The voxel values within the node are encoded by the PPM algorithm
E	**Empty:** The node has no voxel space

To explain the adjustable octree easily, the adjustable quad trees are adopted and used. **Figure AMD1-3** and **Figure AMD1-4** show the examples of the adjustable octrees. In the figures, the white voxels are represented by the white boxes and the white circles with W labels. And the non-white voxels are represented by the color boxes and the circles with B labels. If a node has children of S/W/B/P/E labels, the label of the node is S. If a node is to be decoded by PPM(Prediction by Partial Matching) decoding, the label is P. The empty nodes are represented by the dotted boxes and the dotted circles with E labels. Given a resolution, the full octree nodes and the empty nodes can be recognized and be found out. The decoder need not receive any bitstream or information for empty nodes, because it is possible to know the locations of all empty nodes with only the resolution information. In the adjustable octree, the parent node is subdivided into 8 children nodes regularly as equal as possible. Subdividing a parent node, the subdivision order of 8 children nodes is front left-top, front right-top, front left-bottom, front right-bottom, rear left-top, rear right-top, rear left-bottom, and rear right-bottom. In x, y, z axes, the length of each axis is divided into two sub-parts equally if possible. If it is not possible to divide equally, the length of one sub-part is one voxel longer than the length of the other sub-part. For example, **Figure AMD1-3** shows that it is divided unequally in x-axis as 2 column voxels and 1 column voxel, but it is divided equally in y-axis as 2 row voxels and 2 row voxels.

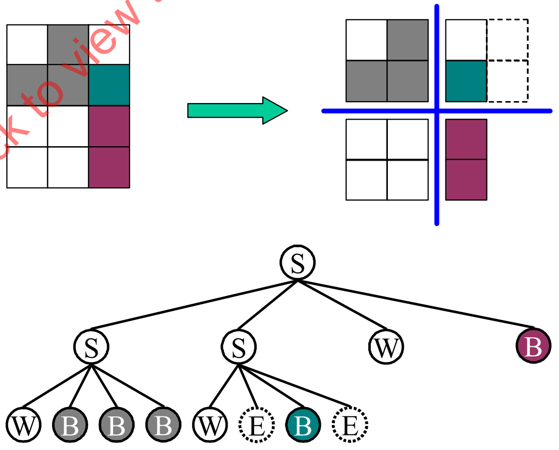

Figure AMD1-3 — Example of adjustable octree in the resolution of 3×4

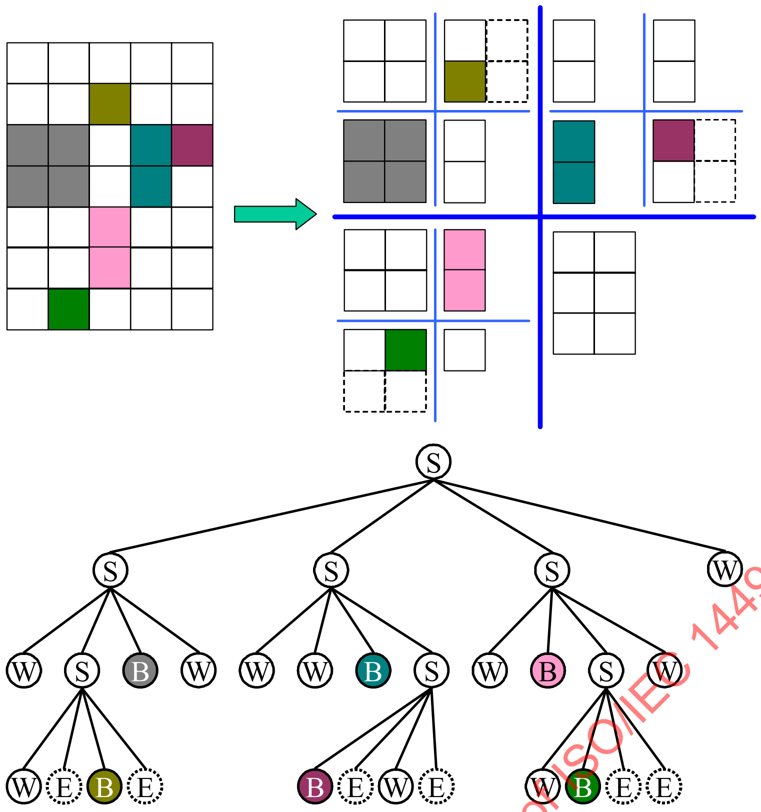

Figure AMD1-4 — Example of adjustable octree in the resolution of 5×7

5.3.2.5.3 Entropy Decoding

The PointTexture decoder receives the bitstream and decodes the header information and the tree node information. To read the bitstream, the context-based adaptive arithmetic decoder is used as the entropy decoder [1]. The bitstream structure for compressed PointTexture is shown in **Figure AMD1-5**.

The number of the tree nodes (nNumberOfTreeNodes) can be obtained by the initalizeOctree function which make the initial full octree nodes with initializing white (0) values excluding the empty nodes in the resolution of nWidth × nHeight × nDepth. If the number of the tree nodes is N, all nodes to be decoded are Node-1, Node-2, Node-3, …, Node-N.

If the nPercentOfDecoding is 100 (maximum value), all nodes will be decoded losslessly. Otherwise, the tree nodes will be decoded as much as nNumberOfTreeNodes × nPercentOfDecoding / 100.

The header information contains the above resolution values and the percent of decoding. Each information of tree node is composed of two parts as SOP and DIB in **Figure AMD1-5** (b). SOP is one bit flag, which indicates that the tree node is split node or ppm node. If the node is split node, the bitstream structure is shown in **Figure AMD1-5** (c). Otherwise, the bitstream structure of ppm node is shown in **Figure AMD1-5** (d).

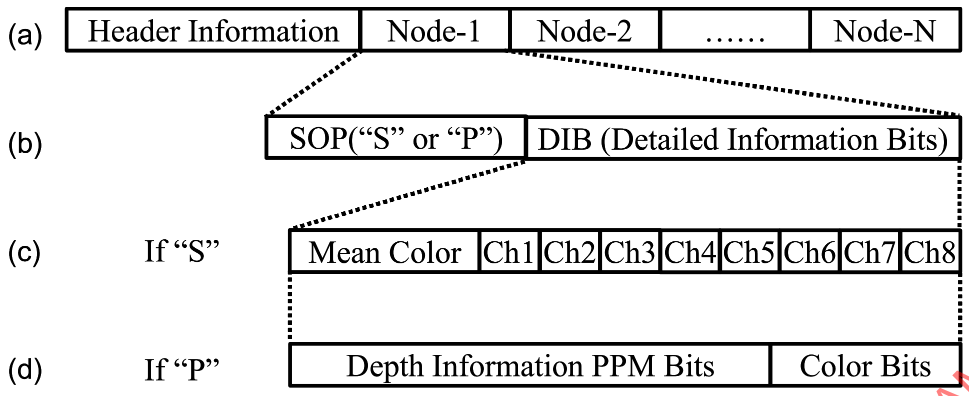

Figure AMD1-5 — Example of the bitstream structure for compressed PointTexture

5.3.2.5.4 Tree Node Decoding

Given an octree resolution, the number of all tree nodes, N can be calculated and obtained. When the decoder received the bitstream of the tree nodes from root node to leaf nodes, it must know the decoding order in the tree nodes. A modified BFS (Breadth First Search) using a priority order queue can be used for the decoding order. If the decoder use a pure original breadth first search, then the progressive decoding is not possible, but the only sequential decoding is possible. So, modified BFS that is a modified BFS using a priority order queue is proposed and adopted to decode and show the progressive PointTexture. In the children's nodes, every first child node of the parent node is higher than the other child node of the parent nodes. Every second child node is higher than the node from the third to the eighth. Every eighth child node is most low than the other child node of the parent node in the priority. According to the children's priority from a parent node, the current decoding node can be notified to the decoder. **Figure AMD1-6** shows an example of the decoding order for the tree nodes in **Figure AMD1-3**. In the figure, the empty E nodes are skipped and ignored in the decoding order. **Figure AMD1-7** shows another example of the decoding order for the tree nodes in **Figure AMD1-4**.

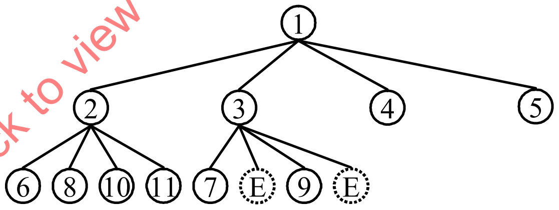

Figure AMD1-6 — Example of the decoding order for the tree nodes in figure AMD1-3

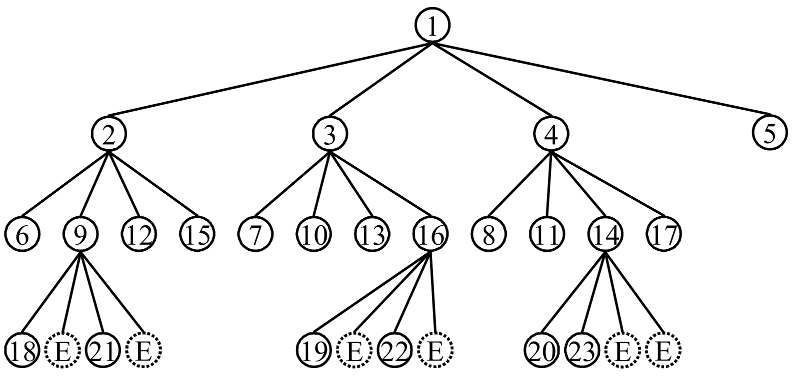

Figure AMD1-7 — Example of the decoding order for the tree nodes in AMD1-4

Figure AMD1-8 shows the block diagram of the tree node decoder. The decoding procedure is as follows. First, the decoder reads the SOP and finds out whether the tree node is split node or ppm node. Second, the decoder reads the DIB(Detailed Information Bits) and finds out where the voxels are and which colors are in each voxel. There are two cases, split node and ppm node. In case of split node, all color values of the children voxels are temporarily set to the average/mean color. These color values are updated when new sub colors of the children are decoded/received. In case of PPM node, the depth information of the voxel region is reconstructed using PPM decoding and the color information is also reconstructed using AAC(Adaptive Arithmetic Coder) decoding and the inverse-DPCM.

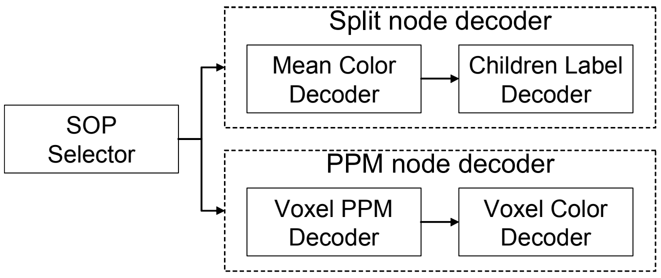

Figure AMD1-8 — Block diagram of tree node decoder

In SOP Selector, the decoder reads the 1 bit flag, which indicates whether the node is the split node or PPM node. If the value is 0, it means split node. Otherwise, it means ppm node. In case of split node, the node is decoded as shown in **Figure AMD1-5**(c). Firstly, the average/mean R, G, B color values are decoded with the adaptive arithmetic decoder. Next, the children labels are also decoded with the adaptive arithmetic decoder. The child label can be black (1) or white (0). If the label is white, all children nodes of the node are white nodes. If the label is black, then the sub children nodes of the label are temporarily regarded as all black nodes. The sub children's nodes and colors can be known in detail when the next sub node of each label is decoded/reached.

In case of PPM node, the node is PPM decoded using the previous decoded voxel values as context. **Figure AMD1-9** is an example of context. Voxels are represented as circles or squares. Gray circles means decoded black voxels and white circles means decoded white voxels. Question marked circles are not decoded voxels. A question marked black square shown in **Figure AMD1-9** (b) is the voxel to be decoded. To decode the voxel as 0 or 1, ten neighboring voxels are used as context excluding three voxels which are marked with 'X'. Similar to the raster scan order, '0' is the previous decoded white circled voxel and '1' is the previous decoded gray circled voxel. The ten bits are used as the context of the squared voxels. In this example, the context of the squared voxels is '0111000011'. Using this context, the black squared voxel is decoded by the context-based adaptive arithmetic decoder.

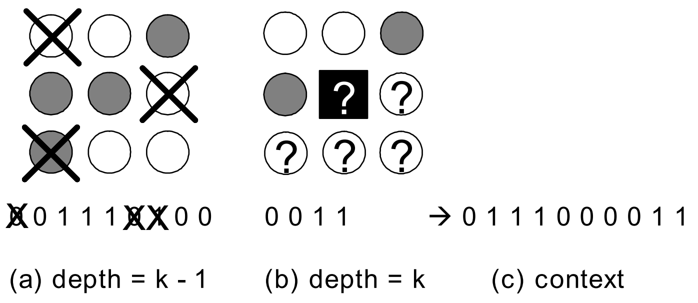

Figure AMD1-9 — Example of context: a voxel represented by a question marked black square in (b) is the voxel to be decoded and voxels represented by white circles and gray circles in (a) and (b) are used to make a context excluding three voxels which are marked with 'X'

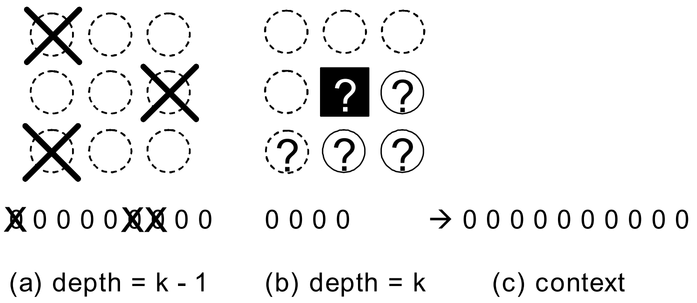

Figure AMD1-10 — Example of context: voxels represented by dotted circles in (a) and (b) are not existed voxels and a voxel to be decoded in (b) is located at a corner actually

After decoding the voxel/depth information, the R, G, B color values of the black voxels within the node are decoded with the adaptive arithmetic decoder and the inverse-DPCM. The inverse-DPCM of color information is performed using the following equations.

nRed = (255 - nDeltaRed) + nRedPrevious , if nDeltaRed > 255

nRed = nDeltaRed + nRedPrevious , if nDeltaRed $\leq$ 255

nGreen = (255 - nDeltaGreen) + nGreenPrevious , if nDeltaGreen > 255

nGreen = nDeltaGreen + nGreenPrevious , if nDeltaGreen $\leq$ 255

nBlue = (255 - nDeltaBlue) + nBluePrevious , if nDeltaBlue > 255

nBlue = nDeltaBlue + nBluePrevious , if nDeltaBlue $\leq$ 255

The values of nDeltaRed, nDeltaGreen, nDeltaBlue are the decoded values after reading and decoding the bitstream of color bits. The values of nRedPrevious, nGreenPrevious, nBluePrevious are the previous decoded values of nRed, nGreen, nBlue. The values of nRed, nGreen, nBlue are the final decoded values which are supposed to be had.

5.3.2.5.5 Adjustable Octree Reconstruction

In the tree node decoding, the labeled adjustable octree reconstruction is also performed. **Figure AMD1-11** shows a simple example of the adjustable octree reconstruction in the resolution of 3×4. **Figure AMD1-11** (b)(d)(f) show the reconstruction process in the decoder side. On the other hand, **Figure AMD1-11** (a)(c)(e) shows the construction process in the encoder side. In **Figure AMD1-11** (b)(d)(f), blue line box means the current decoding node and green line box means the decoding children nodes. In **Figure AMD1-11** (b), the decoding node is split node and the decoding children nodes are B, B, W and B. In **Figure AMD1-11** (d), the decoding node is PPM node and the decoding children nodes are W, B, B and B. In **Figure AMD1-11** (f), the decoding node is split node and the decoding children nodes are W and B. In this case, E nodes are not decoded. They can only be recognized and found by the resolution information.

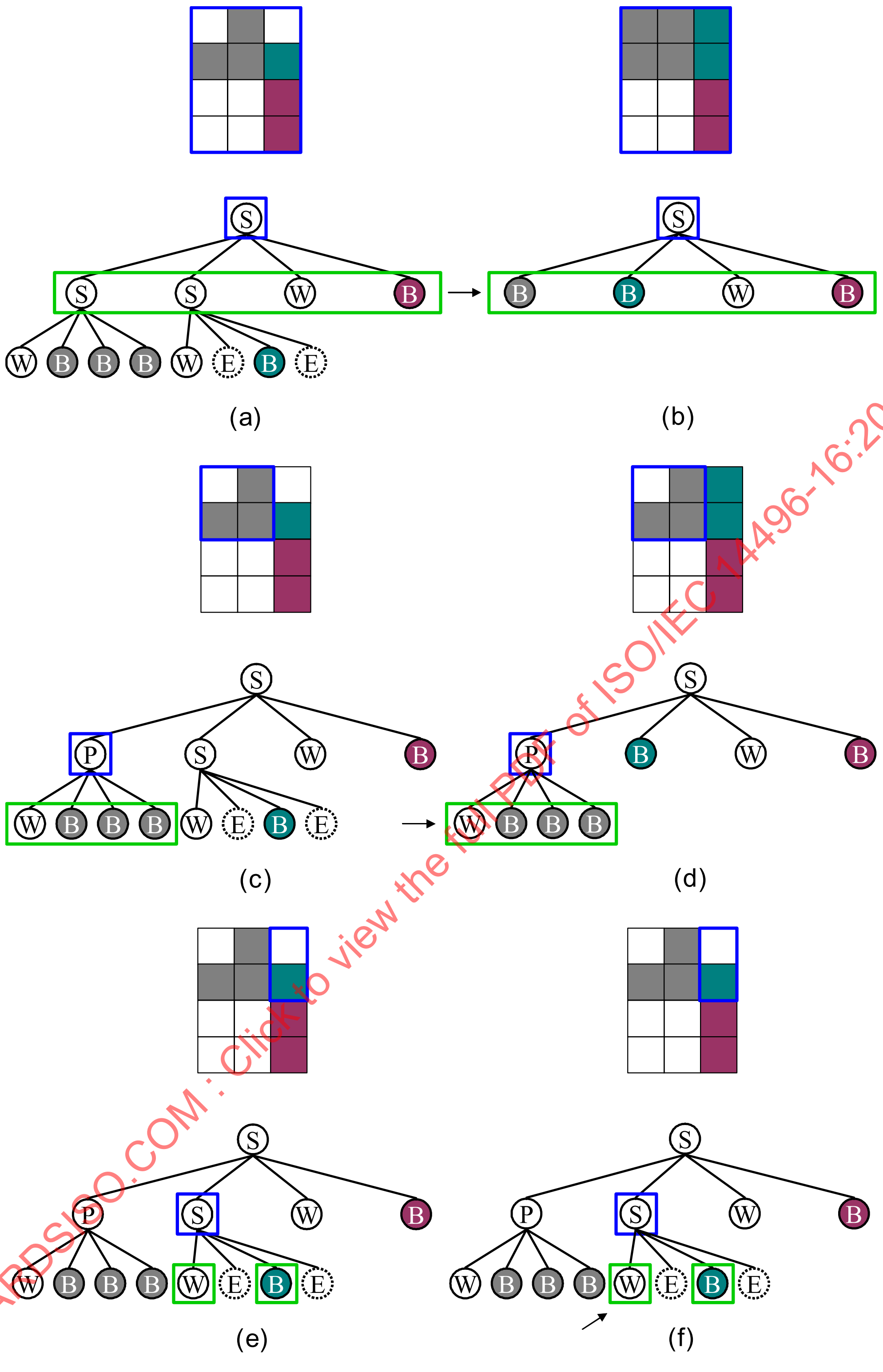

Figure AMD1-11 — Example of the adjustable octree construction/reconstruction

5.3.2.5.6 Voxel Data Reconstruction

After reconstruction of the labeled adjustable octree, it is converted into the voxel data. The resolution of the voxel data is nWidth × nHeight × nDepth. After reconstruction of the voxel data, it can be converted into the PointTexture easily. The PointTexture node has the depth information and the color information to represent the reconstructed 3D objects. With the labeled octree and the efficient bitstream structure, the progressive decoding is possible.

Add clause 5.6 Multiplexing of 3D Compression Streams: syntax of MPEG4 3D Graphics stream:

5.6 Multiplexing of 3D Compression Streams: the MPEG-4 3D Graphics stream (.m3d) syntax

When coded 3D compression objects are carried without MPEG-4 System, the 3D object elementary streams follow the syntax below. The syntax provides for the multiplexing of multiple elementary streams into a single bitstream.

The *3DCObjectSequence* defines container that is used to carry the *3DCObjectSequence* header and the *3DCObject*. The *3DCObjectSequenceHeader* defines the identification of profile and level for this bitstream and the user data defined by users for their specific applications. For example, it can contain scene information for the contained bitstream.

The *3DCObject* defines container that is used to carry the *3DCObject* header and the 3D compressed bitstream: 3D Mesh Compression (3DMC), Interpolator Compression (IC), Wavelet Subdivision Surface (WSS) and Bone-based Animation (BBA). In the *3DCObject* header, the user data also can be defined by users for their specific applications for 3D compression object. The *3DCObject* header contains the "*3dc_object_verid*" which indicates the version number for the tool list of 3D compression object types. And, "*3dc_object_type_start_code*" indicates what 3D compression object type stream is carried and its correspondence decoder. (i.e. "*3dc_object_type_start_code == Simple_3DMC*", the 3DMC decoder decodes the contained bitstream). For other types of 3D compression objects, one container per 3D compression object is used.

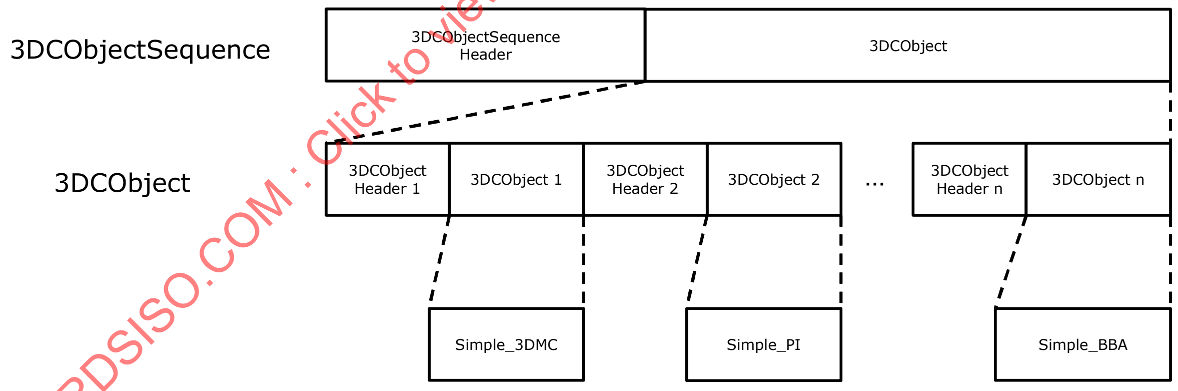

Figure AMD1-12 — Syntax of the MP43D stream

5.6.1 3DCObjectSequence

5.6.1.1 Syntax

```
class 3DCObjectSequence (){
   bit(32)    3dc_object_sequence_start_code;
   bit(8)     profile_and_level_indication;
   bit(32)    *next;
   while (next == user_start_code) {
```

```
            user_data();
    }
    do {
        3DCObject();
        bit(32)    *next;
    }while(next != 3dc_object_sequence_end_code );
  bit(32)   3dc_object_sequence_end_code;
}
```

5.6.1.2 Semantics

3dc_object_sequence_start_code: The 3dc_object_sequence_start_code is the bit string ‘000001A0’ in hexadecimal. It initiates a 3D Compression session.

profile_and_level_indication: This is an 8-bit integer used to signal the profile and level identification. The meaning of the bits is given in Table **AMD1-5**.

Table AMD1-5 — FLC table for profile_and_level_indication

Profile/Level	Code
Reserved	00000000
Core Profile/Level 1	00000001
Core Profile/Level 2	00000010
Reserved	00000011
:	:
Reserved	11111111

3dc_object_sequence_end_code: The 3dc_object_sequence_end_code is the bit string ‘000001A1’ in hexadecimal. It terminates a 3D Compression session.

5.6.2 3DCObject

5.6.2.1 Syntax

```
class 3DCObject(){
    bit(32)    3dc_object_start_code ;
    bit(1) is_3dc_object_identifier;
    bit(3) 3dc_object_verid;
    bit(4) 3dc_object_priority;

    bit(32)    *next;
    while (next == user_start_code) {
            user_data();
      }
    bit(32)    *next;
    if (next == "Simple_3DMC") {
        bit(32)    simple_3dc_object_type_start_code;
        3D_Mesh_Object() ;
      }
    else if (next == "Simple_WSS") {
        bit(32)   simple_WSS_object_type_start_code;
        Wavelet_Mesh_Object();
        }
    else if (next == "Simple_CI") {
        bit(32)   simple_CI_object_type_start_code;
        CompressedCoordinateInterpolator() ;
        }
    else if (next == "Simple_OI") {
        bit(32)   simple_OI_object_type_start_code;
        CompressedOrientationInterpolator () ;
        }
    else if (next == "Simple_PI") {
        bit(32)   simple_PI_object_type_start_code;
        CompressedPositionInterpolator ();
        }
    else if (next == "Simple_BBA") {
        bit(32)   simple_BBA_type_start_code;
        bba_object ();
        }
}
```

5.6.2.2 Semantics

3dc_object_start_code: The 3dc_object_start_code is the bit string '000001A2' in hexadecimal. It initiates a 3D Compression object.

is_3dc_object_identifier: This is a 1-bit code which set to '1' indicates that version identification and priority is specified for the 3D Compression object.

3dc_object_verid: This is a 4-bit code which identifies the version number of the 3D Compression object. Its meaning is defined in Table **AMD1-6**. When this field does not exist, the value of 3dc_object_verid is '0001'.

Table AMD1-6 — Meaning of 3dc_object_verid

3dc_object_verid	Meaning
0000	reserved
0001	Object Types listed in Table AMD1-7
0010-1111	reserved

Table **AMD1-7** list the tools included in each of the Object Types. The current object types can be extended when new tools or functionalities will be introduced.

Table AMD1-7 — Tools for 3D Compression Object Types

AFX Tools	3D Compression Object Types					
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Simple WSS	Simple BBA
3D Mesh Compression (3DMC) - Basic	X					
CoordinateInterpolator (CI)		X				
PositionInterpolator (PI) - Key Preserving - Path Preserving			X			
OrientationInterpolator (OI) - Key Preserving - Path Preserving				X		
Wavelet Subdivision Surface (WSS) - IndexedFaceSet for base mesh					X	
BBA - Only Bones						X

3dc_object_priority: This is a 3-bit code which specifies the priority of the 3D compression object. It takes values between 1 and 7, with 1 representing the highest priority and 7, the lowest priority. The value of zero is reserved.

3dc_object_type_start_code: It is the bit string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' in binary for resynchronization. The last 8 bits represent the one of the values in the 'A6' to 'AB' to indicated object types defined in Table **AMD1-8.** According to the last 8 bits in **"3dc_object_type_start_code"**, the corresponding decoder is called and the compressed stream is decoded. If more object types are defined in Table 3, the added object types are reflected in Table **AMD1-8**.

Table AMD1-8 — Meaning of start code value

3dc_object_type_start_code	code (hexadecimal)
Reserved	A5
Simple 3DMC	A6
Simple CI	A7
Simple PI	A8
Simple OI	A9
Simple WSS	AA
Simple BBA	AB
Reserved	AC through FF

5.6.3 user_data

5.6.3.1 Syntax

```
class user_data()
{
   bit(23) * next;
   while (next != 0) {
      bit(8) user_data_bits;
      bit(23)    * next;
     }
}
```

5.6.3.2 Semantics

user_data_start_code: The user_data_start_code is the bit string ‘000001A4’ in hexadecimal. It identifies the beginning of user data. The user data continues until receipt of another start code.

user_data: This is an 8 bit integer, an arbitrary number of which may follow one another. User data is defined by users for their specific applications. In the series of consecutive user_data bytes there shall not be a string of 23 or more consecutive zero bits.

Add clause 6 AFX object code:

6 AFX Object code

AFX object code	**Object**
0x00	3D Mesh Compression
0x01	WaveletSubdivisionSurface
0x02	MeshGrid
0x03	CoordinateInterpolator
0x04	OrientationInterpolator
0x05	PositionInterpolator
0x06	OctreeImage
0x07	BBA
0x08	PointTexture

Add clause 7, 3D Graphics Profiles:

7 3D Graphics Profiles

7.1 Introduction

A 3D graphics profile defines the set of tools that a product or application compliant with that profile must implement. In MPEG-4 there are defined several profile dimensions. The ones of interest for 3D graphics are "Scene Graph", "Graphics" and "3D Compression" dimensions. The first two refers to nodes in the scene graph and the last refers to compression tools. A profile is defined inside a dimension. One product or application can be compliant with only one profile in a dimension but it can combine several profiles belonging to different dimensions.

7.2 Graphics Dimension

7.2.1 MPEG-4 X3D Interactive Graphics Profiles and Levels

7.2.1.1 Application areas

The application areas are:

- Electronic commerce
- Distance learning
- Instructional Manuals
- Entertainment

7.2.1.2 List of tools/functionalities

The X3D Interactive Graphics profile represents a collection of nodes to allow implementation of a low-footprint engine (e.g. a Java applet or small browser plugin) and is intended to address limitations of software renderers. This set of nodes matches with the nodes related to graphics within the set of nodes being used as an Interactive profile within the X3D standard's development.

The following graphics nodes are supported within this profile: Appearance, Box, Color, Cone, Coordinate, Cylinder, ElevationGrid, IndexedFaceSet, IndexedLineSet, Material, PointSet, Shape, Sphere and TextureCoordinate.

7.2.1.3 Comparison with existing profiles

The X3D Interactive Graphics profile is based on X3D level 1 Interactive Profile [2], and adds the following MPEG-4 features: BIFS-commands for streaming and Quantization for compression efficiency. No other profile in MPEG-4 addresses 3D (only) environments.

7.2.1.4 Supporting Companies, committed to specify Conformance testing

Support was received from the following companies:

- OpenWorlds
- Samsung AIT
- Yumetech
- Web3D Source Group

7.2.1.5 X3D Interactive Graphics Profile @ Level 1 Definition

In the following table, definitions for level 1 of the X3D Interactive Graphics profile are given.

Table AMD1-9 — Level 1 of Web3D Interactive Graphics profile

Node	Minimum System Support
Appearance	*textureTransform* not supported.
Background	*groundAngle* and *groundColor* not supported. *texture* treated as background image FALSE. *stretchToFit* not supported. *backURL*, *frontURL*, *leftURL*, *rightURL*, *topURL* are not supported. One *skyColor*.
Box	Full support

Color	15,000 colors.
Cone	Full support
Coordinate	65,535 points.
Cylinder	Full support
DirectionalLight	*AmbientIntensity* not supported. Not scoped by parent Group or Transform.
IndexedFaceSet	*set_colorIndex* not supported. *set_normalIndex* not supported. *ccw* not supported. *normal* not supported. Only convex indexed face sets supported. Hence, *convex* is not supported. For *creaseAngle*, only 0 and pi radians supported. *normalIndex* not supported. 10 vertices per face. 5000 faces. 65,535 indices in any index field. Face list shall be well-defined as follows: 1. Each face is terminated with -1, including the last face in the array. 2. Each face contains at least three non-coincident vertices. 3. A given *coordIndex* is not repeated in a face. 4. The vertices of a face shall define a planar polygon. 5. The vertices of a face shall not define a self-intersecting polygon.
IndexedLineSet	*set_colorIndex* not supported. *set_coordIndex* not supported. *ccw* not supported. 15,000 total vertices. 15,000 indices in any index field.
Material	*AmbientIntensity* not supported. *shininess* not supported. *SpecularColour* not supported. A Material with emissiveColour not equal to (0,0,0), diffuseColor equal to (0,0,0) is an unlit Material. One-bit transparency; transparency values >= 0.5 transparent.
PointLight	Ignore *radius*. Ignore *ambientIntensity*. Linear attenuation.
PointSet	5000 points.
Shape	Full support.
Sphere	Full support
SpotLight	Ignore *beamWidth*. Ignore *radius*. Ignore *ambientIntensity*. Linear attenuation.
TextureCoordinate	65,535 coordinates.

Table **AMD1-10** specifies further restriction to the fields of the nodes listed in Table **AMD1-9**. These tables can be used for both the Profile and the Level definitions.

Table AMD1-10 — Functionality limitation and minimum system requirement

Node	**Restrictions**
All lights	8 simultaneous lights.
Names for DEF/PROTO/field	50 utf8 octets.
All *url* fields	10 URLs.

SFBool	Full support.
SFColor	Full support.
SFFloat	Full support.
SFImage	256 width. 256 height.
SFInt32	Full support.
SFNode	Full support.
SFRotation	Full support.
SFString	30,000 utf8 octets.
SFTime	Full support.
SFVec2d	15,000 values.
SFVec2f	15,000 values.
SFVec3d	15,000 values.
SFVec3f	15,000 values.
MFColor	15,000 values.
MFFloat	1,000 values.
MFInt32	20,000 values.
MFNode	500 values.
MFRotation	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.
MFVec2d	15,000 values.
MFVec2f	15,000 values.
MFVec3d	15,000 values.
MFVec3f	15,000 values.

Note: The X3D interactive Profile is a common compatibility point with We3D's X3D Interactive Profile. The following tools are supported in MPEG, but not in Web3D's profile: *QuantizationParameter*, *Node Update*, *Scene Update* and *Route Update*. [Note to be included in the standard!]

7.3 Scene Graph Dimension

7.3.1 MPEG-4 X3D Interactive Scene Graph Profile and Levels

7.3.1.1 Application area

The application areas are

- Electronic commerce
- Distance learning
- Instructional Manuals
- Entertainment

7.3.1.2 List of tools/functionalities

The X3D Interactive Scene Graph profile represents a collection of nodes to allow implementation of a low-footprint engine (e.g. a Java applet or small browser plugin) and is intended to address limitations of software renderers. This set of nodes matches with the nodes related to scene description within the set of nodes being used as a Interactive profile within the X3D standard's development.

The following scene graph nodes are supported in this profile: Background, DirectionalLight, PointLight and SpotLight.

7.3.1.3 BIFS nodes for support of Audio and Visual objects

If this profile is used in combination with an Audio and/or a Visual Profile, the required nodes are inferred from these Audio/Video Profiles respectively.

If this profile is used in combination with an Audio Profile: Sound, AudioClip.

If this profile is used in combination with a Visual Profile ImageTexture MovieTexture.

7.3.1.4 Comparison with existing profile

No other profile in MPEG-4 addresses 3D-only interactive environments. The existing 'Complete' profile is a much larger set of tools.

7.3.1.5 Supporting Companies, committed to specify Conformance testing

Support was received from the following companies:

- OpenWorlds
- Samsung AIT
- Yumetech
- Web3D Source Group

7.3.1.6 MPEG-4 X3D Interactive Scene Graph Profile @ Level 1 Definition

In the following table, definitions for level 1 of the X3D Interactive Scene Graph profile are given.

Table AMD1-11 — Level 1 of X3D Interactive Scene Graph profile

Node	Minimum Browser Support
Anchor	*addChildren* not supported. *removeChildren* not supported. Ignore *parameter*.
ColorInterpolator	Full support.
CoordinateInterpolator	15,000 coordinates per *keyValue*.
CylinderSensor	Full support.
Group	*AddChildren* not supported. *removeChildren* not supported.
Inline	Full support.
NavigationInfo	*avatarSize* not supported. *speed* not supported. Ignore *visibilityLimit*
OrientationInterpolator	Full support.
PlaneSensor	Full support.
PositionInterpolator	Full support.
ProximitySensor	Full support.
ScalarInterpolator	Full support.
SphereSensor	Full support.
Switch	Full support.
TimeSensor	Ignored if *cycleInterval* < 0.01 second.
TouchSensor	*hitNormal_changed* not supported
Transform	*addChildren* not supported. *RemoveChildren* not supported.
Viewpoint	Ignore *fieldOfView*.
WorldInfo	Full support

QuantizationParameter	Full support
Node Updates	Full support
Scene Updates	Full support
Route Updates	Full support

The X3D Interactive Graphics profile is based on X3D level 1 Interactive Profile, and adds the following MPEG-4 features: BIFS-commands for streaming and Quantization for compression efficiency.

Table **AMD1-12** specifies other aspects of functionality that are supported by this profile. Note that general items refer only to those specific nodes listed in Table **AMD1-11**.

Table AMD1-12 — Functionality Limitations and Minimum System Requirements

Node	**Minimum System Support**
All groups	500 children. Ignore *bboxCenter* and *bboxSize*.
All interpolators	1000 key-value pairs.
Names for DEF/PROTO/field	50 utf8 octets.
All *url* fields	10 URLs
SFBool	Full support.
SFFloat	Full support.
SFImage	256 width. 256 height.
SFInt32	Full support.
SFNode	Full support.
SFRotation	Full support.
SFString	30,000 utf8 octets.
SFTime	Full support.
SFVec2d	15,000 values.
SFVec2f	15,000 values.
SFVec3d	15,000 values.
SFVec3f	15,000 values.
MFFloat	1,000 values.
MFInt32	20,000 values.
MFNode	500 values.
MFRotation	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.
MFVec2d	15,000 values.
MFVec2f	15,000 values.
MFVec3d	15,000 values.
MFVec3f	15,000 values.

7.4 3D Compression Dimension

The 3D Compression Dimension is defined on the same level as OD, Video, Audio, Graphics, Scene Graph, MPEGJ and Text dimensions and is signalized as indicated in ISO/IEC 14496-1:2005/Amd.1.

3DCompressionProfileLevelIndication is defined in Table AMD1-13.

Table AMD1-13 — 3DCompressionProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Core	L1
0x02	Core	L2
0x0A-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no 3D Compression profile specified	-
0xFF	no 3D Compression capability required	-
Note: Usage of the value 0xFE may indicate that the content described by this descriptor does not comply to any conformance point specified in ISO/IEC 14496-16		

7.4.1 "Core 3D Compression" Profile and Levels

The "Core 3D Compression" profile combines the 3D Mesh Compression, Interpolation Compression, Wavelet Subdivision Surface, and Bone Based Animation tools for efficient 3D resource transmission and storage.

7.4.1.1 Application Areas

- 3D Game
- 3D GUI
- 3D avatar
- 3D background

7.4.1.2 List of Tools/Functionalities

The "Core 3D Compression" profile represents a collection of compression tools to allow implementation of minimum functionalities for compact transmission and storage of 3D object under a constrained environment (e.g. mobile), where the processing power and memory size can be very limited.

The "Core 3D Compression" profile contains the following 3D Compression object types:

- The **Simple 3DMC** object type provides high compression and error resilience for static triangle 3D models.
- The **Simple CI** object type compresses the Coordinate Interpolator animation.
- The **Simple PI** object type compresses the Position Interpolator animation. It can support both Key-Preserving and Path-Preserving mode.
- The **Simple OI** object type compresses the Orientation Interpolator animation. It can support both Key-Preserving and Path-Preserving mode.
- The **Simple WSS** object type represents, in a compressed form, the details for subdivision of 3D mesh. This tool is used for level of detail management and animation. It only allows IndexedFaceSet as a base mesh.
- The **Simple BBA** object type compresses the skeleton animation based on bone transforms and connected to a skin mesh model. This object type does not support Muscle.

Table AMD1-14 — 3D Compression Object Types

AFX Tools	3D Compression Object Types					
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Simple WSS	Simple BBA
3D Mesh Compression (3DMC) - Basic	X					
CoordinateInterpolator (CI)		X				
PositionInterpolator (PI) - Key Preserving - Path Preserving			X			
OrientationInterpolator (OI) - Key Preserving - Path Preserving				X		
Wavelet Subdivision Surface (WSS) - IndexedFaceSet for base mesh					X	
BBA - Only Bones						X

The "Core 3D Compression" includes the object types as illustrated in Table AMD1-14.

Table AMD1-15 — "Core 3D Compression" Profile

	"3D Compression" Object Types					
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Simple WSS	Simple BBA
Core 3D Compression Profile	X	X	X	X	X	X

7.4.1.3 Comparison with Existing Profiles and object types

The Core 3D Compression Profile is the first profile defined in the "3D Compression" profile dimension.

7.4.1.4 Supporting Companies

- Samsung Electronics: http://www.samsung.com/
- Electronics Telecommunication Research Institute: http://www.etri.re.kr/
- Gomid: http://www.gomid.com/
- Reakosys: http://www.reakosys.com/
- VirtualDigm: http://www.vdigm.com/
- Nexus Chips: http://www.nexuschips.com
- Geosoft: http://geosoft.co.kr/
- Wow4M: http://www.wow4m.com/

7.4.1.5 Profile Level Definition

According to target device and applications, we defined two levels as listed in **Table AMD1-16, AMD1-17**, and **AMD1-18**. The level 1 is for the mobile device without H/W graphics accelerator. Thus, it is suitable simple application such as 3D Background, 3D GUI, 3D Pre-Viewer and 3D Avatar. On the other hand, the level 2 is for the mobile device supported by H/W graphics accelerator. Thus, the level 2 is suitable for the 3D Game application in addition to the application in level 1.

Table AMD1-16 — Levels and data constraints

Level	Data Constraints
Level 1	- Number of triangles in a scene ≤ 500 - Number of objects in a scene ≤ 30 - Number of bones attachable to an object ≤ 10 - Number of bones affecting a vertex ≤ 2 - Number of vertex in Coordinate Interpolator = 0
Level 2	- Number of triangles in a scene ≤ 5000 - Number of objects in a scene ≤ 100 - Number of bones attachable to an object ≤ 30 - Number of bones affecting a vertex ≤ 4 - Number of vertex in Coordinate Interpolator > 0

Table AMD1-17 — Levels and functionalities constraints

Level	Functionality Constraints
Level 1	- Triangle only - colorIndex and texCoordIndex not supported (coordIndex values are used instead) - The vertices of a face shall not define a self-intersecting polygon - A given coordIndex is not repeated in a face - Ignore normal and normalIndex - The color and texCoord cannot be supported at the same time. - Only base mesh supported
Level 2	- Triangle only - Full support of colorIndex, normalIndex, texcoordIndex - The vertices of a face shall not define a self-intersecting polygon - A given coordIndex is not repeated in a face - Color and Texture can be blended. - Only base mesh supported

Table AMD1-18 — Levels and player constraints

Level	Player Constraints (Informative Recommendation)
Level 1	- Size of Texture memory ≤ 1 M - Number of light ≤ 1 - Directional Light - Material: ambientIntensity, diffuseColor, emissiveColor - Transparency - Flat and Gouraud shading - Texture Blending not supported
Level 2	- Size of Texture memory ≤ 4 M - Number of light ≤ 8 - Directional Light, PointLight and SpotLight - Material: ambientIntensity, diffuseColor, emissiveColor, shininess, specularColor - Transparency - Flat and Gouraud shading - Texture Blending - Texture Filtration - Perspective correction - Total Decoding Time ≤ 10sec

Add clause 8, XMT for AFX tools:

8 AFX nodes

8.1 AFX nodes

The XMT syntax for all the nodes defined in 14496-16 is specified in the xsd file attached to ISO/IEC 14496-11.

8.2 AFX encoding hints

8.2.1 WaveletSubdivision encoding hints

8.2.1.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.2.1.2 Semantics

The <BitWrapperWaveletSubdivisionEncodingHints> allow the source and/or target formats to be described so that when encoding is required the author can control various parameters of the encoding process where supported by the encoder.

8.2.2 MeshGrid encoding hints

8.2.2.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.2.2.2 Semantics

The <BitWrapperMeshGridEncodingHints> allow the source and/or target formats to be described so that when encoding is required the author can control various parameters of the encoding process where supported by the encoder.

8.2.3 OctreeImage encoding hints

8.2.3.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.2.3.2 Semantics

The <BitWrapperOctreeImageEncodingHints> allow the source and/or target formats to be described so that when encoding is required the author can control various parameters of the encoding process where supported by the encoder.

8.2.4 PointTexture encoding hints

8.2.4.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.2.4.2 Semantics

The <BitWrapperPointTextureEncodingHints> allow the source and/or target formats to be described so that when encoding is required the author can control various parameters of the encoding process where supported by the encoder.

8.2.5 BBA encoding hints

8.2.5.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.2.5.2 Semantics

The <BBAEncodingHints> allow the source and/or target formats to be described so that when encoding is required the author can control various parameters of the encoding process where supported by the encoder.

8.3 AFX encoding parameters

8.3.1 WaveletSubdivisionEncodingParameters

8.3.1.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.3.1.2 Semantics

NbBpSC: stores the number of bits on which to encode the scaling coefficients minus one.

NbBPX: stores the number of bits on which to encode the first component of the waelet coefficients.

NbBPY: stores the number of bits on which to encode the second component of the waelet coefficients.

NbBPZ: stores the number of bits on which to encode the third component of the waelet coefficients.

Wtype: stores the type of subdivision scheme used in the low pass filtering.

lift: indicates whether the high pass filtering is lifted or not. This is an integer type.

isInLocalCoordinates: indicates whether the wavelet coefficients are expressed in local or global coordinates.

LengthNbBits: indicates the number of bits used to represent the number of bits used to encode the length of the bitplanes. This is an integer type.

isPartial: indicates whether the transmission is in full mode or bitplane by bitplane. This is an integer type.

8.3.2 MeshGridEncodingParameters

8.3.2.1 Syntax

The syntax is described in the xsd file attached to ISO/IEC 14496-11.

8.3.2.2 Semantics

nLevels: specifies the maximum number of resolution levels (in the {u,v,w} directions of the reference-grid) of the encoded model. If no value is specified, the encoded model will preserver the number of resolution levels of the original MeshGrid model.

nSlices: specifies the maximum number of slices (in the {u,v,w} directions of the reference-grid) corresponding to the last resolution level of the encoded model. If no value is specified, the encoded model will preserver the number of slices of the original MeshGrid model.

sizeROI: specifies the size of the region of interest (ROI), and is defined with respect to the number slices of the first resolution level of the model. It is used to split the original model into ROIs when encoding. The parameter **modeROI** specifies how the value of **sizeROI** is adapted for the higher resolution levels of the model. If no value is specified for **sizeROI**, the model is encoded at each resolution level as one entity.

modeROI: may yield one of the following two values: “constantDensity” and “constantSize”. When its value is equal to “constantDensity” it means that at each resolution level the size of the ROI should be chosen such that the number of reference-grid points within the ROI remains approximately the same. This implies that the value of **sizeROI** (with respect to the reference-grid) is kept constant for each resolution level. When the value of **modeROI** is “constantSize”, then the size of the ROI with respect to the model should stay the same at each resolution level, hence the value of **sizeROI** is duplicated with each higher resolution level.

groupROI: specifies the number of ROIs that are grouped together (for efficiency reasons) when encoding the reference-grid points. It is only used for encoding the reference-grid points, the mesh is still encoded on a ROI basis.

hasConnectivityInfo: specifies if true that the connectivity should be encoded. If no value is specified, the value from the original mesh is considered.

hasRefineInfo: specifies if true that the vertices’ offsets should be encoded. If no value is specified, the value from the original mesh is considered.

hasRepositionInfo: specifies if true that the vertices’ inter-level refinement information should be encoded. If no value is specified, the value from the original mesh is considered.