INTERNATIONAL STANDARD

ISO/IEC 14496-16

Second edition
2006-12-15

AMENDMENT 1
2007-10-01

Information technology — Coding of audio-visual objects —

Part 16: Animation Framework eXtension (AFX)

AMENDMENT 1: Geometry and shadow

Technologies de l'information — Codage des objets audiovisuels —

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

AMENDEMENT 1: Géométrie et ombre

Reference number
ISO/IEC 14496-16:2006/Amd.1:2007(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 1 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*.

Information technology — Coding of audio-visual objects —

Part 16:
Animation Framework eXtension (AFX)

AMENDMENT 1: Geometry and shadow

Add subclause 4.3.6 MultiResolution FootPrint-Based Representation:

4.3.6 MultiResolution FootPrint-Based Representation

4.3.6.1 Introduction

MultiResolution FootPrint-Based representation is a solution to represent any set of objects based on footprints (a set of IndexedLineSet, or for a near future, buildings, cartoons…). The main interests in this representation are its progressivity, view dependency, and compression.

4.3.6.2 FootPrintSetNode

4.3.6.2.1 Node Interface

FootPrintSetNode { #NDT=%SFGeometryNode

exposedField	**MFGeometryNode**	children	[]

}

4.3.6.2.2 Functionality and semantics

The **children** field specifies the list of all footprints rendered according to the current viewpoint. This list contains currently FootPrintNode representing the set of footprints rendered from the current viewpoint. This list can be updated at each displacement of the viewpoint in order to adapt the scene complexity to the view. This representation can be extended to be used with any object based on footprints such as buildings, cartoons, etc. In this case, the children field can contain BuildingPartNode to represent buildings.

4.3.6.3 FootPrintNode

4.3.6.3.1 Node Interface

FootPrintNode { #NDT=%SFGeometryNode

exposedField	**SFInteger**	index	-1
exposedField	**SFIndexLineSet2D**	footprint	NULL

}

4.3.6.3.2 Functionality and semantics

Index: this is the index of the node corresponding to a footprint elevation at a specific level of detail. This index is essential for streaming, due to the synchronization between the representation on the server and on the client. This index will be sent to the server as a refinement request.

Footprint: this is an IndexLineSet2D describing the footprint.

4.3.6.4 BuildingPartPrintNode

4.3.6.4.1 Node Interface

BuildingPartNode { #NDT=%SFGeometryNode

exposedField	**SFInteger**	index	-1
exposedField	**SFIndexLineSet2D**	footprint	NULL
exposedField	**SFUnsigned integer**	buildingIndex	-1
exposedField	**SFFloat**	height	0
exposedField	**SFFloat**	altitude	0
exposedField	**MFGeometryNode**	alternativeGeometry	[]
exposedField	**MFRoofNode**	roofs	[]
exposedField	**MFFacadeNode**	facades	[]

}

4.3.6.4.2 Functionality and semantics

Index: this is the index of the node corresponding to a footprint elevation at a specific level of detail. This index is essential for streaming, due to the synchronization between the representation on the server and on the client (this index will be sent to the server as a refinement request).

Footprint: this is a IndexLineSet2D describing the footprint.

buildingIndex: this is the index of the building to which this part is connected. A building corresponds to a group of building parts having the same buildingIndex.

Height: this is the height of the building.

Altitude: this is the altitude of the building (corresponding to the floor of the prism).

alternativeGeometry: this is a geometry node corresponding to an optional object used to replace the normal building. This alternative geometry can be used to swap a building with a more detailed model (used for example to replace a footprint elevation based model of a monument, by a more detailed model). In this case, the footprint-based elevation model will not be rendered, since the alternative model will be.

roofs: this is a node array allowing to describe complete roofs that will be reconstructed on top of the footprint elevation.

facades: this is a node array allowing to describe in detail the modelling of the façades corresponding to this building part. The size of this array corresponds to the number of facades, equivalent to the number of edges of the polygon defining the footprint.

4.3.6.5 RoofNode

4.3.6.5.1 Node Interface

RoofNode { #NDT=%SFGeometryNode

exposedField	**SFInteger**	type	0
exposedField	**SFFloat**	height	0.0
exposedField	**MFFloat**	slopeAngle	[0.0]
exposedField	**SFFloat**	eaveProjection	0.0
exposedField	**SFInt**	edgeSupportIndex	-1
exposedField	**SFURL**	roofTextureURL	""
exposedField	**SFBool**	isGenericTexture	TRUE
exposedField	**SFFloat**	textureXScale	1.0
exposedField	**SFFloat**	textureYScale	1.0
exposedField	**SFFloat**	textureXPosition	0.0
exposedField	**SFFloat**	textureYPosition	0.0
exposedField	**SFFloat**	textureRotation	0.0

}

4.3.6.5.2 Functionality and semantics

type: this is the type of the roof. 0 – Flat Roof, 1 – Symmetric Hip Roof, 2 – Gable Roof, 3 – Salt Box roof, 4 – Non Symmetric Hip Roof.

height: this is the height of the roof that allows cropping it. (This is not used for flat roofs).

slopeAngle: this is the angle of the roof slopes in degrees (useless for flat roofs). In the case of a Symmetric Hip Roof, all slopes have the same angle. In the case of a Non Symmetric Hip Roof, each slope has a specific angle.

eaveProjection: this is the projection of the eave (useless for flat roofs).

edgeSupportIndex: this is the index of the edge in the footprint that supports the roof (use only for Salt Box roofs)

roofTextureURL: this is the URL of the texture that is orthogonally mapped onto the roof

isGenericTexture: this specifies whether the texture mapped onto the roof is generic or not. In the case of a generic texture, the reference system is centred on the top left vertex of the roof pan, and axed perpendicularly to the gutter. In the case of an aerial photograph, the reference system is centred on the first vertex of the footprint, and axed on the world coordinate system.

textureXScale: this is the scaling of the roof texture along X-axis

textureYScale: this is the scaling of the roof texture along Y-axis

textureXPosition: this is the displacement of the texture along X-axis

textureYPosition: this is the displacement of the texture along Y-axis

textureRotation: this is an angle in radian specifying the rotation to apply to the texture.

4.3.6.6 FacadefNode

4.3.6.6.1 Node Interface

FacadeNode { #NDT=%SFGeometryNode
}

exposedField	**SFFloat**	WidthRatio	1.0
exposedField	**SFFloat**	XScale	1.0
exposedField	**SFFloat**	YScale	1.0
exposedField	**SFFloat**	XPosition	0.0
exposedField	**SFFloat**	YPosition	0.0
exposedField	**SFFloat**	XRepeatInterval	0.0
exposedField	**SFFloat**	YRepeatInterval	0.0
exposedField	**SFBool**	Repeat	FALSE
exposedField	**SFURL**	FacadePrimitive	""
exposedField	**SFInteger**	NbStories	0
exposedField	**MFInteger**	NbFacadeCellsByStorey	0
exposedField	**MFFloat**	StoreyHeight	1.0
exposedField	**MFFacadeNode**	FacadeCellsArray	[]

4.3.6.6.2 Functionality and semantics

WidthRatio: this corresponds to a ratio between the width of the cell compared to the width of the parent cells.

XScale: this is a parameter allowing scaling in X-coordinate the model corresponding to the URL Façade Primitive (2D texture or 3D model). For texture, this scale corresponds to the real size in X-coordinate in meters of the texture. For 3D Model, this size corresponds to the scale to apply on the model in X-coordinate.

NOTE This scale is very important as the model can be used for different buildings, and must be adjusted to the current one.

YScale is a parameter allowing scaling in Y-coordinate the model (2D texture or 3D model). For texture, this scale corresponds to the real size in Y-coordinate in meters of the texture. For 3D Model, this size corresponds to the scale to apply on the model in Y-coordinate.

NOTE This scale is very important as the model can be used for different buildings, and must be adjusted to the current one.

XPosition: this is a parameter allowing moving in X-coordinate the model in the cell defined by the FacadePrimitiveArray of the father node. This position can be essential to place a primitive in the centre of the cell.

YPosition: this is a parameter allowing moving in Y-coordinate the model in the cell defined by the FacadePrimitiveArray of the father node. This position can be essential to place a primitive in the center of the cell.

Repeat: this is a Boolean that is TRUE if and only if the model has to be repeated all over the cell defined by the father node.

NOTE This is essential for texture mapping, or to repeat regularly a model of windows all over a façade.

FacadePrimitive: this is a link to the corresponding primitive (Texture or 3D model) that have to be mapped onto the cell.

NbStories: this is the number of stories of the façade.

NbFacadeCellsByStorey: this is an array that defines the number of cells by storey. This parameter is essential to know on which storey corresponds a cell in FaçadeCellsArray.

StoriesHeight: this is an array specifying the height of each storey.

FacadeCellsArray: this is an array of FacadeNode that links each cell to a facadeNode (another array of cells, and/or a façade primitive like a texture or a 3D model). The size of this array is the sum of all NbFacadeCellsByStorey[i] , for all i from 0 to NbStories.

Add subclause 4.7.3 BBA Animation Algorithm:

4.7.3 BBA Animation algorithm

The initial pose of an articulated model must contain a skeleton that is aligned with the mesh. Thus some bones have a non-identity initial transformation. During the animation, the bone transforms are updated. Since the skeleton and the mesh are originally aligned, only the offset between the new bone transforms and the initial ones has to be applied to the vertices.

Animating the skinned model consists then in the following steps:

a) for all bones compute the initial transformation in the local space as the combination of the elementary transform: rotation, translation, center, scale and scaleOrientation; all these components are expressed in the parent's coordinate system.

b) for all the bones, compute the initial transformation in the world space as a product between the initial transformation of the bone in the local space and the initial transformation of the bone's parent expressed in the world space

c) compute the inverse of the previous transformation

d) at each animation frame, update the local elementary transforms: rotation, translation, center, scale and scaleOrientation

e) at each animation frame, repeat step b)

f) at each animation frame, for all the bones multiply the transformation obtained at step e) with the one computed at step c)

g) at each animation frame, for each pair bone/vertex, multiply the vertex with the transform obtained at the previous step and with the corresponding weight.

Add Subclause 4.8 Scene tools:

4.8 Scene tools

4.8.1 Shadows

The **Shadow** node works as a special grouping node for the author defined creation of hard and soft shadows caused by 3D-surfaces, shadow properties and **SpotLight** nodes.

4.8.1.1 Syntax

Shadow {			
eventIn	MFNode	**addChildren**	
eventIn	MFNode	**removeChildren**	
exposedField	MFNode	**children**	[]
exposedField	SFBool	**enabled**	TRUE
exposedField	MFBool	**cast**	TRUE

exposedField	MFBool	**receive**	TRUE
exposedField	SFFloat	**penumbra**	0

}

4.8.1.2 Semantics

addChildren: the addChildren event appends nodes to the grouping node's children field. Any nodes passed to the addChildren event that are already in the group's children list are ignored.

removeChildren: the removeChildren event removes nodes from the grouping node's children field. Any nodes in the removeChildren event that are not in the grouping node's children list are ignored.

children: contains a list of children nodes. The **children** node's surfaces are generally invisibly rendered. Only instances of 3D-surfaces are able to work as occluders and receivers for shadow creation in association only with **SpotLight** nodes. Each **children**[***m***] and its descendants correspond to the combination of shadow properties **cast**[***m***] and **receive**[***m***]. If it is intended, that a 3D-surface has to work as occluder or receiver, it must fulfil several prerequisites. The assigned children has to be a single instance 3D-surface or an instance 3D-surface that is part of a sub-graph. A **SpotLight** must be associated to this 3D-surface in the same way as shown in Figure AMD1-2. The light source has to illuminate the 3D-surface, for casting or receiving shadows.

enabled: the functionality of the **Shadow** node is enabled with the value TRUE. The functionality of the **Shadow** node is disabled with the value FALSE.

cast: assigns the capability to a 3D-surface to cast shadows onto other 3D-surfaces. With the value TRUE a single instance or a branch with instances of 3D-surfaces included becomes an occluder. The field works as **MFBool,** so every **children**[***m***] (single node or branch) is able to have its own value of **cast**. The shadow properties of a 3D-surface's node instance are transmitted according the ID of MediaObject to all of those instances of 3D-surfaces existing outside of **Shadow** nodes in that scene with the same ID (see Figure AMD1-2).

receive: assigns the capability to a 3D-surface to receive shadows from itself or from other surfaces. With TRUE a single instance or a branch with instances of 3D-surfaces included becomes a receiver. The field works as **MFBool,** so every **children**[***m***] (single node or branch) is able to have its own value of **receive**. The shadow properties of a 3D-surface's node instance are transmitted according the ID of MediaObject to all of those instances of 3D-surfaces existing outside of **Shadow** nodes in that scene with the same ID (see Figure AMD1-2). The field works as **MFBool,** so every **children**[***m***] node is able to have its own value of **receive**. The shadow properties of a 3D-surface's node instance become transmitted to all of those instances of 3D-surfaces existing outside of **Shadow** nodes in that scene (see Figure AMD1-3).

penumbra: describes the geometrical extension of the related **SpotLight** as a sphere radius.

4.8.1.3 Annex

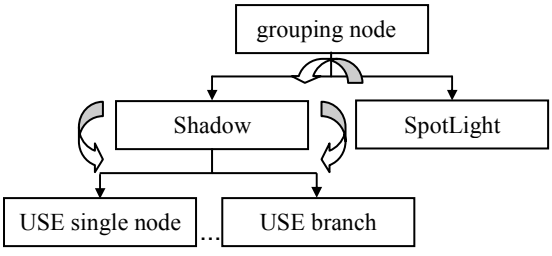

Figure AMD1-1 — Semantical representation

If **Shadow** and **SpotLight** nodes own the same grouping node as parent, a shadow relationship is created automatically between them.

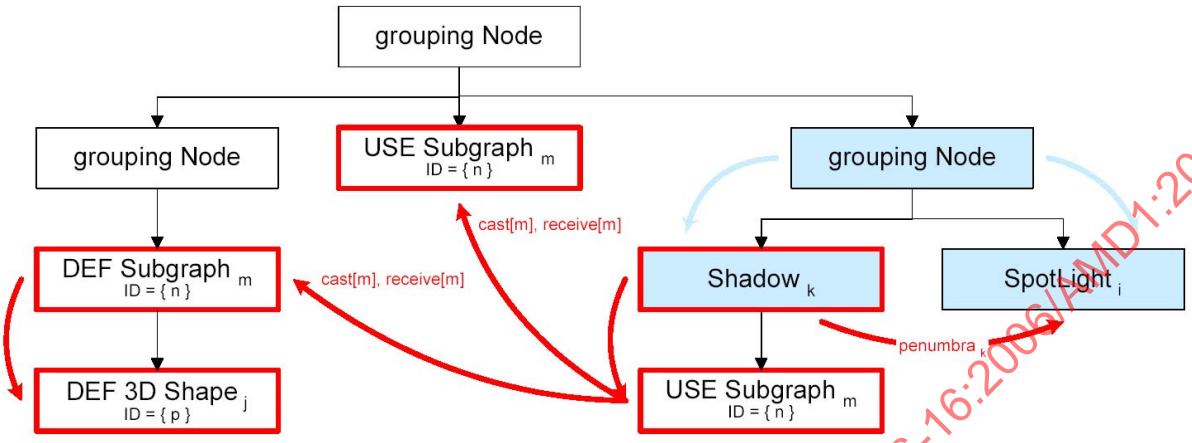

Figure AMD1-2 — Transmission of shadow properties to 3D-surfaces and light sources

The combination of multiple **Shadow** nodes with one or multiple **SpotLight** nodes possesses several shadow properties associated with one **SpotLight** node. This way a **SpotLight** node gets several **penumbra** values (see Figure AMD1-3). Additionally it can create multiple shadow properties with a single **Shape** node.

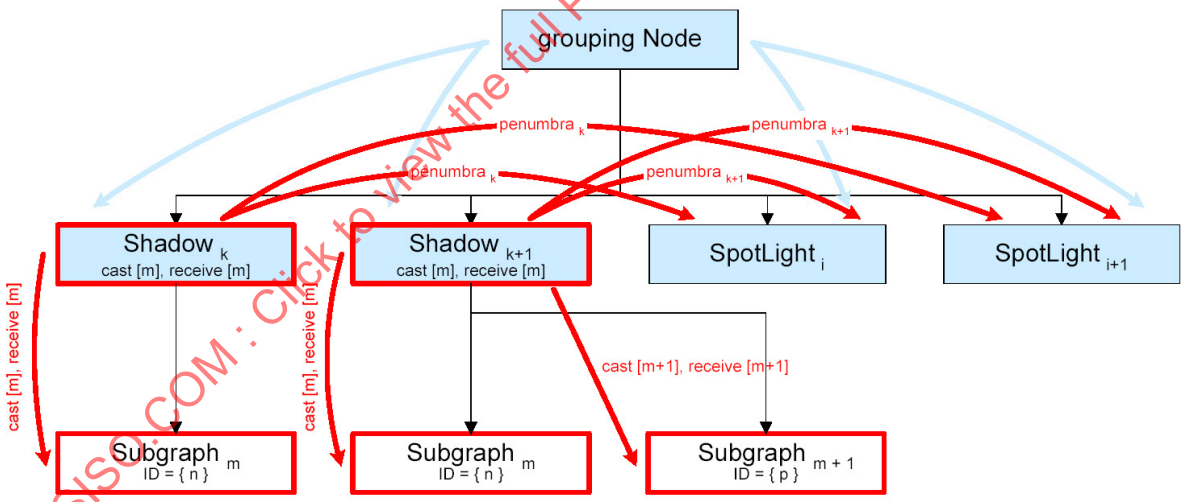

Figure AMD1-3 — Transmission of shadow properties to multiple 3D-surfaces and light sources

The following rules were formulated for those cases. If a unique 3D-surface is related to the same light source several times, the shadow properties are handled by the Boolean operation OR. All **penumbra** values add up and increase the light body extension.

The occurrence of multiple associated **SpotLight** nodes combined with only one **Shadow** node simply results in each **SpotLight** node creating another independent relation.

Add subclause 5.8 MultiResolution FootPrint-Based Representation:

5.8 MultiResolution FootPrint-Based Representation

5.8.1 Downstream syntax

This is the syntax of the downstream MultiResolution FootPrint-based Representation.

5.8.1.1 FootPrintSetDecoderSpecificInfo

5.8.1.1.1 Syntax

```
class FootPrintsDecoderConfig extends AFXDecoderSpecificInfo {
SDLInt<16>    FPObjectType
SDLInt<32>    MaxNbFootPrints
SDLInt<6>     FootPrintNbBits
SDLFloat      Step
SDLInt<6>     NbBitsMetricError
SDLFloat      MinX
SDLFloat      MaxX
SDLFloat      MinY
SDLFloat      MaxY
SDLInt<1>     DEFIdUsed
switch (FPObjectType)
   {
      case 1 :   FPBuildingDecoderConfig FPBuildingDSI
   }
}
```

5.8.1.1.2 Semantics

FPObjectType: This is an integer specifying the type of the multiresolution footprint-based representation (0 for classic footprints, but extended types could be considered.)

MaxNbFootPrints: this is the number of footprints in the footprint-based representation.

FootPrintNbBits: this is the number of bits used to decode the footprint indices. Its value is the lowest integer superior or equal to to $\log_2(\texttt{MaxNbFootprints})$.

Step: this is the smallest spatial subdivision.

NbBitsMetricError: this is the number of bits on which to encode the metric error (32 or 64 bits)

MinX: this is the minimum X-coordinate of the model

MaxX: this is the maximum X-coordinate of the model

MinY: this is the minimum Y-coordinate of the model

MaxY: this is the maximum Y-coordinate of the model

DEFIDUsed: If **DEFIDUsed** is **TRUE**, then the ID used during the Bifs encoding is used as reference. Otherwise, the string defname is used.

NOTE For future extension, depending of the object type (buildings, cartoons…) other parameters could be added to this decoder configuration. For current simple footprints, these parameters are enough to configure the decoder.

5.8.1.2 FPBuildingDecoderSpecificInfo

5.8.1.2.1 Syntax

```
class FPBuildingDecoderConfig {
SDLFloat   MinAltitude;
SDLFloat   MaxHeight;
SDLInt<6>  NbBitsZBuilding;
SDLInt<6>  NbBitsNbStories;
SDLInt<6>  NbBitsStoreyHeight;
SDLInt<6>  NbBitsFacadeWidth;
SDLInt<6>  NbBitsNbFacadeCellsByStorey;
}
```

5.8.1.2.2 Semantics

MinAltitude: this is the minimum altitude of the set of footprint-based elevations.

MaxHeight: this is the maximum height of the set of footprint-based elevations.

NbBitsZBuilding: this is the number of bits used to encode the altitude and height of buildings.

NbBitsNbStories: this is the number of bits used to specify the number of stories by façade element

NbBitsStoreyHeight: this is the number of bits used to specify the height ratio of each storey.

NbBitsFacadeWidth: this is the number of bits used to specify the width ratio of each façade element.

NbBitsNbFacadeCellsByStorey: this is the number of bits used to specify the number of cells per storey.

5.8.1.3 FootPrintSet Message

The FootPrintSet Message is intended to carry all the set base and refinement information for the design of footprint sets.

5.8.1.3.1 Syntax

```
FootPrintSetMessage {
   int(32) NbFootPrints
   For (int i=0; i<NbFootprints; i++)
      FootPrintMessage   FootPrint;
}
```

5.8.1.3.2 semantics

NbFootprints: this is an integer giving the number of FootPrintMessage that have to be read in the stream.

5.8.1.4 FootPrint Message

A Footprint message is intended to carry a base or refinement information for the design of footprint sets.

5.8.1.4.1 Syntax

```
class FootprintMessage {
   int(FPNbBits) index
   bit(1) type
   FPNewVertices FPNV
   int(6) IndexNbBits
   if (type) {
```

```
      int(10) offspring
      for (i=0; i<offspring; i++) {
         int(FPNbBits) localIndex
         float(NbBitsMetricError) MetricError
         IndexFootprintSet IFPS
         switch (FPObjectType)
         {
            case 1: FPBuildingParameters FPBP
         }
      }
   }
   else
   {
      float(NbBitsMetricError) MetricError
      int(8) NbRings
      For (int i=0; i<NbRings-1; i++)
      {
         int(IndexNbBits) FirstVertexIndex
         switch (FPObjectType)
         {
            case 1: FPBuildingParameters FPBP
         }
      }
   }
}
```

5.8.1.4.2 Semantics

MetricError: this is the geometric error between the original model and the simplified model used by the client to decide if this node has to be refined.

IndexNbBits: this is the number of bits used to decode the vertices indexes. Its value is the lowest integer superior or equal to $\log_2$(`FootPrintsDecoderConfig.MaxIndex`).

Index: this is the index identifying the current footprint.

Type: this is a Boolean with value 0 if the current message describes a primary footprint, and 1 if this is a refinement.

FPNV: this is a class describing the new vertices used to refine the current footprint.

Offspring: this is the number of children of the current footprint.

localIndex: this is the index identifying the i-th child of the current footprint.

IFPS: this is a class listing the indices of vertices of the footprint.

NbRings: this is the number of rings in the new footprint.

FirstVertexIndex: this is the index in the new vertices array of the first vertex for each ring (there is no index for the first ring, since it is always equal to 0).

FPBP: this is a class describing the parameters corresponding to the new building based on footprint.

5.8.1.5 FPNewVertices

5.8.1.5.1 Syntax

```
class FPNewVertices
{
   int(6) coordtype
```

```
   int(16) nbNewVertices
   for (i=0; i<nbnewvertices; i++)
   {
      if (type == 0 || step == -1.0)
      {
         float(32)     DeltaX
         float(32)     DeltaY
      }
      else
      {
         bool                        SignDeltaX
         unsigned int(coordtype-1)   AbsDeltaX
         bool                        SignDeltaY
         unsigned int(coordtype-1)   AbsdeltaY
      }
   }
}
```

5.8.1.5.2 Semantics

coordType: this is the number of bits to encode the vertex coordinates.

nbNewVertices: this is the number of vertices described in the rest of the class.

DeltaX, DeltaY: these are the 2D coordinates of the newly added vertex,

SignDeltaX, SignDeltaY: these specifiy whether deltaX and deltaY are positive or not.

AbsdeltaX, AbsdeltaY: these are the 2D absolute value coordinates of the newly added vertex, expressed in a reference system based on the barycentre of the parent footprint vertices. The actual position of the new vertex is obtained by multiplying **AbsDeltax*SignDeltaX** by **Step** (defined in the DecoderSpecificInfo), and adding the coordinates of the barycentre of the parent footprint vertices.

The decoding process is exposed in Annex J.

5.8.1.6 IndexFootprintSet

5.8.1.6.1 Syntax

```
class IndexFootprintSet
{
   int(16)  nbVertexIndices
   for (int i=0; i<nbVertexIndices; i++)
   {
      int(IndexNbBits) index
   }
}
```

5.8.1.6.2 Semantics

nbVertexIndices: this is the number of indices in the rest of the class. **nbVertexIndices=nbVertices** in the footprint + Nulber of rings-1.

index: this is the index of the i-th vertex. If index=-1, a new ring starts.

5.8.1.7 FPBuildingParameters

5.8.1.7.1 Syntax

```
class FPBuildingParameters
{
   int(FPNbBits) buildingIndex
   if (step!=-1.0)
   {
      int(NbBitsZBuilding) altitude
      int(NbBitsZBuilding) height
   }
   else
   {
      float(32) altitude
      float(32) height
   }
   For (int i=0; i<nbFacades; i++)
   {
      FPFacade facades
   }
   int(6) nbRoofs
   for (int i=0; i<nbRoofs; i++)
   {
      FPRoof roof
   }

   int(8) nbSwapNodes
   for (int i=0; i<NbSwapNodes; i++)
   {
      if (DEFIDUse)
         int(32) nodeId
      else
      {
         int(8) nbChar
         for (int j=0; j<nbChar; j++)
         char defname[j]
      }
   }
}
```

5.8.1.7.2 Semantics

buildingIndex: this is the index of the building to which this part is connected.

altitude: if **step** is different from -1.0, the altitude of the building is encoded using a integer. The actual altitude is given by

Decoded=altitude*Step.

height: if **step** is different from -1.0, the height of the building is encoded using a integer. The actual height is gicven by:

DecodedHeight=height*Step.

textureURL: this is the URL of the texture to be applied on the side of the footprint-based elevation.

nbSwapNodes: this is the number of nodes in the scene graph used to swap the block corresponding to the footprint-based elevation.

NodeId: this is the index of a node in the scene graph used to swap the block corresponding to the current footprint-based elevation for a more defined model.

NOTE This footprint-based model swap for a more detailed model is essential if one needs to add a good detailed model that can be represent using a footprint-based representation. For example, one might need to replace the footprint-based model of specific buildings like monuments (used as reference points for the user during the navigation) by a more detailed model that can be representing by a footprint-based elevation.

nbRoofs: this is the number of roofs that are superimposed on top of the footprint elevation.

roof: this is the description of each roof that are superimposed on top of the footprint elevation.

nbfacades: this is the number of facades of the building, equivalent to the number of edges of the polygon defining by the footprint, i.e. equal to `nbVertexIndices`.

facades: this is the description of each facade model that are mapped on each face of the footprint elevation.

5.8.1.8 RoofBitStream

5.8.1.8.1 Syntax

```
class FPRoof {
   int(3) roofType
   switch (roofType)
   {
      case 0 : //Flat Roof

      case 1 : // Symmetric Hip Roof
         float roofHeight
         float roofSlopeAngle
         float roofEaveProjection
      case 2 : // Gable Roof
         float roofHeight
         float roofSlopeAngle
         float roofEaveProjection
      case 3 : // Salt Box Roof
         float roofHeight
         float roofSlopeAngle
         float roofEaveProjection
         int(indexNbBits) roofEdgeSupportIndex
      case 4 : // Non Symmetric Hip Roof
         float roofHeight
         float roofSlopeAngle[nbWalls]
         float roofEaveProjection

   }
   if (DEFIDUse)
      int(32) nodeIdAppearance
   else
   {
      int(8) nbChar
      for (int j=0; j<nbChar; j++)
         char defnameAppearance [j]
   }

   bool   IsGeneric
   int(2) projectionTextureMode
   switch (projectionTextureMode) {
      case 0 : // Generic metric texture

      case 1 : // Generic texture
         float XScale
         float YScale
```

```
      case 2 : // Real texture
         float XScale
         float YScale
         float XPosition
         float YPosition
         float rotation
         float roofEaveProjection
   }
}
```

5.8.1.8.2 Semantics

roofType: this is the type of the roof : 0 – Flat; 1 – Symmetric Hip; 2 – Gable; 3 – Salt Box; 4 – Non Symmetric Hip

roofHeight: this is the height of the roof. If -1.0, the height is not defined; else, the roof is cropped.

roofSlopeAngle: this is the angle of the roof slopes.

roofEaveProjection: this is the length of the roof eave projection

roofEdgeSupportIndex: this is the index of the edge of the polygon defining the footprint elevation that supports the roof.

nodeIDAppearance or defnameAppearance: this is a reference to a Appearance node corresponding tof the texture that is orthogonally mapped on the roof. **nodeIdAppearance** is used if DEFIDUsed is true (see FootprintDSI) else **defnameAppearance**

IsGeneric this specifies whether the texture mapped on the roof is generic (value 1), or obtained from an aerial photograph (value 0). If the roof texture is generic, the reference system is centred on the bottom left vertex of each roof pan, and aligned on the gutter. In the case of a non-generic texture, the reference system is centred on the first vertex of the footprint, and aligned along the world coordinate system.

projectionTextureMode: this is the mode used to map the texture on the roof. It specifies whether the different parameters XScale, YScale, XPosition, YPosition, and rotation are used.

XScale: this is the scaling the roof texture along X-axis

YScale: this is the scaling the roof texture along Y-axis

XPosition: this is the displacement of the texture along X-axis

YPosition: this is the displacement of the texture along Y-axis

rotation: this is an angle in radian specifying the rotation to apply to the texture.

5.8.1.9 FacadeBitStream

5.8.1.9.1 Syntax

```
class FPFacade
{
   int(NbBitsFacadeWidth) WidthRatio
   int(j2)  MappingMode
   Switch (MappingMode)
   {
      Case 0:
```

```
      Case 1:
         float(32) XScale
         float(32) YScale
      Case 2:
         float(32) XScale
         float(32) YScale
         float(32) XPosition
         float(32) YPosition
      Case 3:
         float(32) XScale
         float(32) YScale
         float(32) XPosition
         float(32) YPosition
         float(32) XRepeatInterval
         float(32) YRepeatInterval
   }
   Bool Repeat
   int(2) FacadePrimitiveType
   if (DEFIDUse)
      int(32) nodeIdFacadePrimitiveNode
   else
   {
      int(8) nbChar
      for (int j=0; j<nbChar; j++)
      char defnameFacadePrimitiveNode [j]
   }

   int(NbBitsNbStories) NbStories
   For (int i=0; i<NbStories; i++)
   {
      float(NbBitsStoreyHeight) StoreyHeight
   }
   For (int i=0; i<NbStories; i++)
   {
      int(NbBitsNbFacadeCellsByStorey) NbFacadeCellsByStorey
      For (int j=0; j< NbFacadeCellsByStorey; j++)
      {
         FPFacade FacadeCell
      }
   }
}
```

5.8.1.9.2 Semantics

`XScale`: this is the scale on X-coordinate applied over the model defined by FacadePrimitiveModel

`YScale`: this is the scale on Y-coordinate applied over the model defined by FacadePrimitiveModel

XPosition: this is the translation on X-coordinate applied over the model defined by FacadePrimitiveModel

YPosition: this is the translation on Y-coordinate applied over the model defined by FacadePrimitiveModel

Repeat: if this field has value 1 the texture or model on the façade is repeated.

FacadePrimitiveType: specifies the type of primitive:

0: nothing

1: texture

2: 3Dmodel.

nodeIdFacadePrimitiveNode or defnameFacadePrimitiveNode: this is a rerefence to the node in the scene graph corresponding to the Facade primitive. nodeIdFacadePrimitiveNode is used if DEFIDUsed is true (see FootprintDSI) else defnameFacadePrimitiveNode

NbStories: this is the number of stories of the facade.

NbFacadeCellsByStorey: this is the number of cells for the corresponding storey.

StoreyHeight is the height of the corresponding storey.

NbFacadeCells: this is the number of cells on the façade. It's equal to the sum of NbFacadeCellsByStorey for each storey.

StoreyWidth: this is the width in meter of the corresponding cell.

FacadeCell: this is a façade node corresponding to the cell.

5.8.2 Upstream syntax (for backchannel)

When specified as an upstream in corresponding ES descriptor, the MultiResolutionFootprint Set stream has to be read according to the AFX Generic Backchannel syntax (see subclause 5.5).

Add subclause 5.9 3DMC extension tools:

5.9 3DMC extension tools

3DMC extension tools are basically based on 3D mesh coding (3DMC) tools introduced in MPEG-4 Visual [ISO/IEC 14496-2]. Compared to 3DMC tools, 3DMC extension tools incorporate vertex order and face order preserving functionality, efficient texture mapping functionality, and new stitching operation and remove forest split operation, computational graceful degradation (CGD), and existing stitching operation.

5.9.1 Introduction

3D Mesh Object

The 3D Mesh Object is a 3D polygonal model that can be represented as an IndexedFaceSet in BIFS. It is defined by the position of its vertices (geometry), by the association between each face and its sustaining vertices (connectivity), and optionally by colours, normals, and texture coordinates (properties). Properties do not affect the 3D geometry, but influence the way the model is shaded. 3D mesh coding (3DMC) extension addresses the efficient coding of 3D mesh object. It comprises a basic method and several options. The basic 3DMC extension method operates on manifold model and features incremental representation of single resolution 3D model. The model may be triangular or polygonal – the latter are triangulated for coding purposes and are fully recovered in the decoder. Options include: (a) support for error resilience; (b) vertex order and face order preserving; (c) efficient texture mapping; and (d) support for non-manifold and non-orientable model. The compression of application-specific geometry streams (Face Animation Parameters) and generalized animation parameters (BIFS Anim) are currently addressed elsewhere in this part of ISO/IEC 14496.

In 3DMC extension, the compression of the connectivity of the 3D mesh (e.g. how edges, faces, and vertices relate) is lossless, whereas the compression of the other attributes (such as vertex coordinates, normals, colours, and texture coordinates) may be lossy.

Single Resolution Mode

The incremental representation of a single resolution 3D model is based on the Topological Surgery scheme. For manifold triangular 3D meshes, the Topological Surgery representation decomposes the connectivity of each *connected component* into a *simple polygon* and a *vertex graph*. All the triangular faces of the 3D mesh are connected in the simple polygon forming a *triangle tree*, which is a spanning tree in the dual graph of the 3D mesh. Figure AMD1-4 shows an example of a triangular 3D mesh, its dual graph, and a triangle tree. The vertex graph identifies which pairs of boundary edges of the simple polygon are associated with each other to reconstruct the connectivity of the 3D mesh. The triangle tree does not fully describe the triangulation of the simple polygon. The missing information is recorded as a *marching edge*.

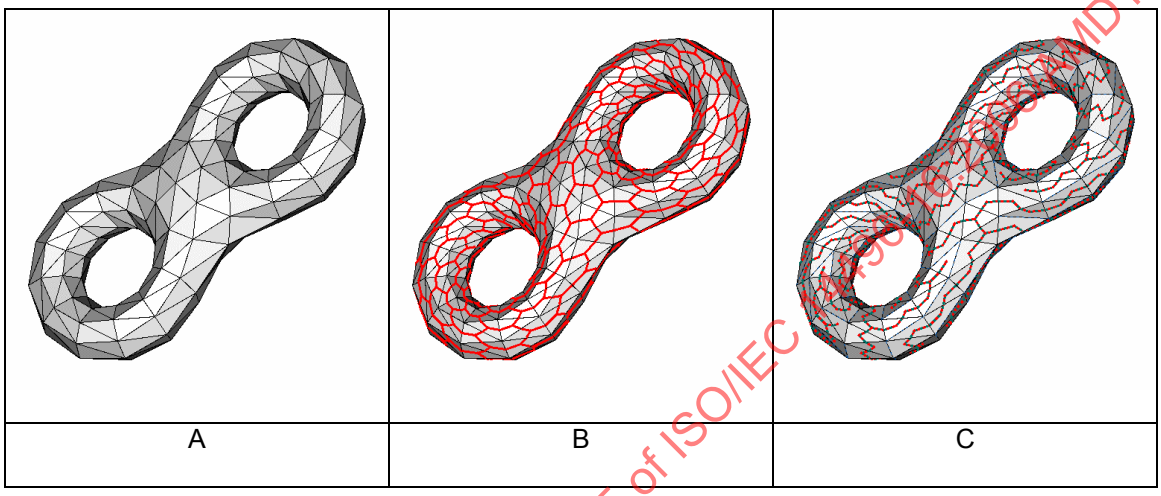

Figure AMD1-4 — A triangular 3D mesh (A), its dual graph (B), and a triangle tree (C)

For manifold 3D meshes, the connectivity is represented in a similar fashion. The polygonal faces of the 3D mesh are connected in a simple polygon forming a *face tree*. The faces are triangulated, and which edges of the resulting triangular 3D mesh are edges of the original 3D mesh is recorded as a sequence of *polygon_edge* bits. The face tree is also a spanning tree in the dual graph of the 3D mesh, and the vertex graph is always composed of edges of the original 3D mesh.

The vertex coordinates and optional properties of the 3D mesh (normals, colours, and texture coordinates) are quantised, predicted as a function of decoded ancestors with respect to the order of traversal, and the errors are entropy encoded.

Incremental Representation

When a 3D mesh is downloaded over networks with limited bandwidth (e.g. PSTN), it may be desired to begin decoding and rendering the 3D mesh before it has all been received. Moreover, content providers may wish to control such incremental representation to present the most important data first. The basic 3DMC method supports this by interleaving the data such that each triangle may be reconstructed as it is received. Incremental representation is also facilitated by the options of partitioning for error resilience.

Error Resilience for 3D Mesh Object

If the 3D mesh is partitioned into independent parts, it may be possible to perform more efficient data transmission in an error-prone environment, e.g. an IP network or datacasting service in a broadcast TV network. It must be possible to resynchronize after a channel error, and continue data transmission and rendering from that point instead of starting over from scratch. Even with the presence of channel errors, the decoder can start decoding and rendering from the next partition that is received intact from the channel.

Flexible partitioning methods can be used to organize the data, such that it fits the underlying network packet structure more closely, and overhead is reduced to the minimum. To allow flexible partitioning, several connected components may be merged into one partition, where as a large connected component may be divided into several independent partitions. Merging and dividing of connected components using different partition types can be done at any point in the 3D mesh object.

Vertex Order and Face Order Preserving

To animate or edit the content represented by IFS, one can do the operation per vertex. The authoring tool is assumed to use a fixed vertex order in a scene for easy and efficient handling of animation and updating. When 3DMC is used on IndexedFaceSet, this presumed order is broken for the IndexedFaceSet node. The encoder of 3DMC changes the vertex order to maximize the compression efficiency. This causes an additional problem when used with other tools that share a fixed vertex order. Not only 3DMC changes vertex order after compression, it also changes the face order. This may not be a problem if editing or animation of a 3D model is done per vertex, where vertex order is a problem to fix in such a case. However, if editing or animation is done per face, the change of face order may have the same impact as vertex order.

These vertex and face order changes may create a lot of confusion not only at the encoder side, but more at the decoder side. Hence, in order to solve this issue it needs to carry original vertex and face order information with encoded bitstream and re-order the vertex and face order accordingly after decoding the encoded model.

Efficient Texture Mapping

Efficient texture mapping would alleviate the need of having very accurate geometry model, since approximation by texture map will do the trick for the user. Therefore, the accuracy of texture coordinates is critical in order to guarantee the quality of rendered quality of 3D models.

Near lossless or lossless compression of texture coordinate, hence, is a very important issue to make sure. The current IndexedFaceSet-based representation describes the texture coordinates in float, where the texture coordinates in reality are discrete values in integer. To compress the texture coordinates losslessly from the point of integer values, two kinds of schemes can be used: (1) if the texture image size is previously known, quantised step size for texture coordinates can be set as the inverse of the texture image size; (2) if the size of texture image is not known, the possible quantised step size can be estimated by analyzing the difference values of the real texture coordinate values (or the regular intervals of ordered texture coordinate values).

Stitching for Non-Manifold and Non-Orientable Meshes

The connectivity of a non-manifold and non-orientable 3D mesh is represented as a manifold 3D mesh and a sequence of *stitches*. Each stitch describes the number of duplications for the vertex increase or face increase and the actual index of the original vertex or face and duplicated vertex (vertices) or face (faces) during the conversion of non-manifold and non-orientable into an oriented-manifold 3D mesh and a sequence of stitches.

Encoder and Decoder Block Diagrams

High level block diagrams of a general 3D polygonal model encoder and decoder are shown in Figure AMD1-5. They consist of a 3D mesh connectivity (de)coder, geometry (de)coder, property (de)coder, vertex/face order (de)coder, and entropy (de)coding blocks. Connectivity, vertex position, and property information are extracted from 3D mesh model described in VRML or MPEG-4 BIFS format. The connectivity (de)coder is used for an efficient representation of the association between each face and its sustaining vertices. The geometry (de)coder is used for a lossy or lossless compression of vertex coordinates. The property (de)coder is used for a lossy or lossless compression of colour, normal, and texture coordinate data. The vertex/face order (de)coder is used for vertex order and face order preserving.

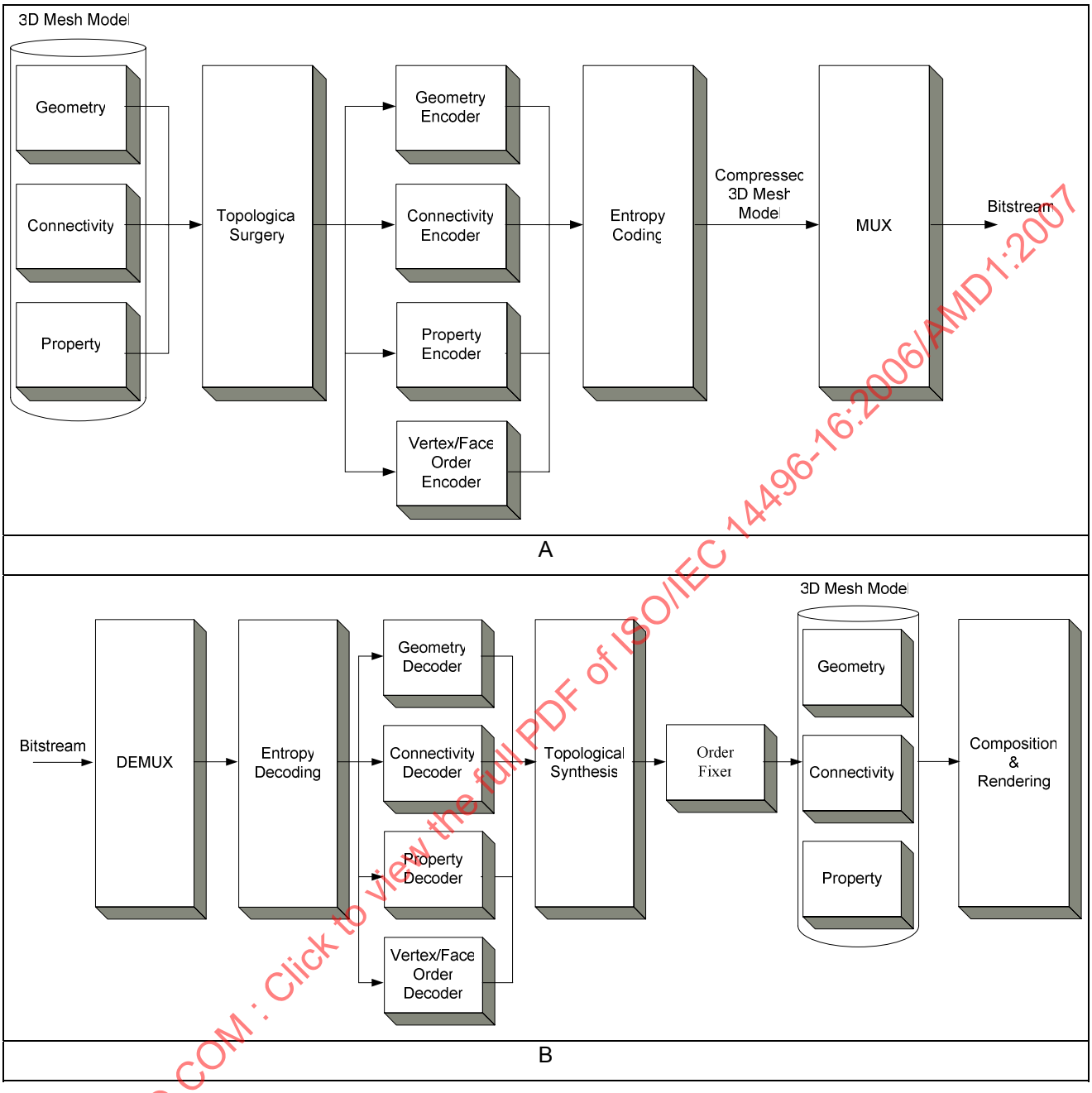

Figure AMD1-5 — General block diagram of the 3D mesh compression.

A: 3D mesh encoder.

B: 3D mesh decoder.

5.9.2 Visual bitstream syntax and semantics

5.9.2.1 Structure of coded visual data

5.9.2.1.1 3D Mesh Object

The compressed bitstream for a 3D mesh is composed of a header data block with global information, followed by a sequence of connected component data blocks, each one associated with one connected component of the 3D mesh.

3D Mesh Header	CC Data #1	...	CC Data #nCC

nCC is the number of connected components.

If a 3D mesh is coded in error resilience mode, connected component data blocks are grouped or divided into partitions.

Partition #1	Partition #2	...	Partition #nPT

Additionally, if the vertex and face order preserving is supported, the last connected component data block is followed by one or two data blocks, each one of them representing vertex order and face order information. Vertex order and face order information can be applied to all the components of a bitstream.

Vertex Order	Face Order

Each connected component data block is composed of three records, the Vertex Graph record, the Triangle Tree record, and the Triangle Data record.

Vertex Graph	Triangle Tree	Triangle Data

The triangle tree record contains the structure of a triangle spanning tree which links all the triangles of the corresponding connected component forming a simple polygon. The 3D mesh is represented in a triangulated form in the bitstream, which also contains the information necessary to reconstruct the original faces. The vertex graph record contains the information necessary to stitch pairs of boundary edges of the simple polygon to reconstruct the original connectivity, not only within the current connected component, but also to previously decoded connected components. The connectivity information is categorized as *global* information (per connected component) and *local* information (per triangle). The global information is stored in the Vertex Graph and Triangle Tree records. The local information is stored in the Triangle Data record. The triangle data is arranged on a per triangle basis, where the ordering of the triangles is determined by the traversal of the triangle tree.

Data for triangle #1	Data for triangle #2	...	Data for triangle #nT

The data for a given triangle is organized as follows:

marching edge	td_orientation	polygon_edge	coord	normal	color	texCoord

The *marching edge, td_orientation* and *polygon_edge* constitute the per triangle connectivity information. The other fields contain information to reconstruct the vertex coordinates (*coord*) and optionally, normal, color, and texture coordinate (*texCoord*) information.

5.9.2.2 Visual bitstream syntax

5.9.2.2.1 3D Mesh Object

5.9.2.2.1.1 3D_Mesh_Object

3D_Mesh_Object () {	No. of bits	Mnemonic
3D_MO_start_code	16	uimsbf
3D_Mesh_Object_Header()		
do {		
3D_Mesh_Object_Layer()		
} while (nextbits_bytealigned() == 3D_MOL_start_code)		
}		

5.9.2.2.1.2 3D_Mesh_Object_Header

3D_Mesh_Object_Header() {	No. of bits	Mnemonic
ccw	1	bslbf
convex	1	bslbf
solid	1	bslbf
creaseAngle	6	uimsbf
coord_header()		
normal_header()		
color_header()		
texCoord_header()		
3DMC_extension	1	bslbf
if (3DMC_extension == '1')		
3DMC_extension_header()		
}		

5.9.2.2.1.3 3D_Mesh_Object_Layer

3D_Mesh_Object_Layer () {	No. of bits	Mnemonic
3D_MOL_start_code	16	uimsbf
mol_id	8	uimsbf
if (mol_id == 0)		
3D_Mesh_Object_Base_Layer()		
else		
3D_Mesh_Object_Extension_Layer()		
}		

5.9.2.2.1.4 3D_Mesh_Object_Base_Layer

3D_Mesh_Object_Base_Layer() {	No. of bits	Mnemonic
do {		
3D_MOBL_start_code	16	uimsbf
mobl_id	8	uimsbf
while (!bytealigned())		
one_bit	1	bslbf
qf_start()		
if (3D_MOBL_start_code == "partition_type_0") {		
do {		
connected_component()		
qf_decode(**last_component**, last_component_context)		vlclbf
} while (last_component == '0')		
}		
else if (3D_MOBL_start_code == "partition_type_1") {		
vg_number=0		
do {		
vertex_graph()		
vg_number++		
qf_decode(has_stitches, has_stitches_context)		vlclbf
qf_decode(**codap_last_vg**, codap_last_vg_context)		vlclbf
} while (codap_last_vg == '0')		
}		
else if (3D_MOBL_start_code == "partition_type_2") {		
if(vg_number > 1)		
qf_decode(**codap_vg_id**)		vlclbf
qf_decode(**codap_left_bloop_idx**)		vlclbf
qf_decode(**codap_right_bloop_idx**)		vlclbf
qf_decode(**codap_bdry_pred**)		vlclbf
triangle_tree()		
triangle_data()		
}		
} while (nextbits_bytealigned() == 3D_MOBL_start_code)		
if (has_stitches)		
stitching()		
}		

5.9.2.2.1.5 coord_header

coord_header() {	No. of bits	Mnemonic
coord_binding	2	uimsbf
coord_bbox	1	bslbf
if (coord_bbox == '1') {		
coord_xmin	32	bslbf

coord_ymin	32	bslbf
coord_zmin	32	bslbf
coord_size	32	bslbf
}		
coord_quant	5	uimsbf
coord_pred_type	2	uimsbf
if (coord_pred_type==”tree_prediction” || coord_pred_type==’”parallelogram_prediction”) {		
coord_nlambda	2	uimsbf
for (i=1; i<coord_nlambda; i++)		
coord_lambda	4-27	simsbf
}		
}		

5.9.2.2.1.6 normal_header

normal_header() {	No. of bits	Mnemonic
normal_binding	2	uimsbf
if (normal_binding != “not_bound”) {		
normal_bbox	1	bslbf
normal_quant	5	uimsbf
normal_pred_type	2	uimsbf
if (normal_pred_type==”tree_prediction”|| normal_pred_type==”parallelogram_prediction”) {		
normal_nlambda	2	uimsbf
for (i=1; i<normal_nlambda; i++)		
normal_lambda	3-17	simsbf
}		
}		
}		

5.9.2.2.1.7 color_header

color_header() {	No. of bits	Mnemonic
color_binding	2	uimsbf
if (color_binding != “not_bound”) {		
color_bbox	1	bslbf
if (color_bbox == ‘1’) {		
color_rmin	32	bslbf
color_gmin	32	bslbf
color_bmin	32	bslbf
color_size	32	bslbf
}		
color_quant	5	uimsbf
color_pred_type	2	uimsbf

if (color_pred_type==”tree_prediction” ‖ color_pred_type==”parallelogram_prediction”) {		
color_nlambda	2	uimsbf
for (i=1; i<color_nlambda; i++)		
color_lambda	4-19	simsbf
}		
}		
}		

5.9.2.2.1.8 texCoord_header

texCoord_header() {	**No. of bits**	**Mnemonic**
texCoord_binding	2	uimsbf
if (texCoord_binding != “not_bound”) {		
texCoord_bbox	1	bslbf
if (texCoord_bbox == ‘1’) {		
texCoord_umin	32	bslbf
texCoord_vmin	32	bslbf
texCoord_size	32	bslbf
}		
texCoord_quant	5	uimsbf
texCoord_pred_type	2	uimsbf
if (texCoord_pred_type==”tree_prediction” ‖ texCoord_pred_type==”parallelogram_prediction”) {		
texCoord_nlambda	2	uimsbf
for (i=1; i<texCoord_nlambda; i++)		
texCoord_lambda	4-19	simsbf
}		
}		
}		

5.9.2.2.1.9 3DMC_extension_header

3DMC_extension_header() {	**No. of bits**	**Mnemonic**
do {		
function_type	4	uimsbf
if (function_type == “Order_mode”)		
Order_mode_header ()		
else if (function_type == “Adaptive_quant_texCoord_mode”)		
Adaptive_quant_texCoord_mode_header()		
} while(function_type != “Escape_mode”)		
}		

5.9.2.2.1.10 Order_mode_header

Order_mode_header () {	No. of bits	Mnemonic
vertex_order_flag	1	bslbf
if(vertex_order_flag)		
vertex_order_per_CC_flag	1	bslbf
face_order_flag	1	bslbf
if(face_order_flag)		
face_order_per_CC_flag	1	bslbf
}		

5.9.2.2.1.11 Adaptive_quant_texCoord_mode_header

Adaptive_quant_texCoord_mode_header () {	No. of bits	Mnemonic
texCoord_quant_u	16	uimsbf
texCoord_quant_v	16	uimsbf
}		

5.9.2.2.1.12 connected_component

connected_component() {	No. of bits	Mnemonic
vertex_graph()		
qf_decode(has_stitches, has_stitches_context)		vlclbf
triangle_tree()		
triangle_data()		
}		

5.9.2.2.1.13 vertex_graph

vertex_graph() {	No. of bits	Mnemonic
qf_decode(**vg_simple**, vg_simple_context)		vlclbf
depth = 0		
code_last = '1'		
openloops = 0		
do {		
do {		
if (code_last == '1') {		
qf_decode(**vg_last**, vg_last_context)		vlclbf
if (openloops > 0) {		
qf_decode(**vg_forward_run**, vg_forward_run_context)		vlclbf
if (vg_forward_run == '0') {		
openloops--		
if (openloops > 0)		
qf_decode(**vg_loop_index**, vg_loop_index_context)		vlclbf
break		

} }		
}		
qf_decode(**vg_run_length**, vg_run_length_context)		vlclbf
qf_decode(**vg_leaf**, vg_leaf_context)		vlclbf
if (vg_leaf == '1' && vg_simple == '0') {		
qf_decode(**vg_loop**, vg_loop_context)		vlclbf
if (vg_loop == '1')		
openloops++		
}		
} while (0)		
if (vg_leaf == '1' && vg_last == '1' && code_last == '1')		
depth--		
if (vg_leaf == '0' && (vg_last == '0' || code_last == '0'))		
depth++		
code_last = vg_leaf		
} while (depth >= 0)		
}		

5.9.2.2.1.14 triangle_tree

triangle_tree() {	No. of bits	Mnemonic
depth = 0		
ntriangles = 0		
branch_position = -2		
do {		
qf_decode(**tt_run_length**, tt_run_length_context)		vlclbf
ntriangles += tt_run_length		
qf_decode(**tt_leaf**, tt_leaf_context)		vlclbf
if (tt_leaf == '1') {		
depth--		
}		
else {		
branch_position = ntriangles		
depth++		
}		
} while (depth >= 0)		
if (3D_MOBL_start_code == "partition_type_2")		
if(codap_right_bloop_idx – codap_left_bloop_idx – 1 > ntriangles) {		
if (branch_position == ntriangles – 2) {		
qf_decode(**codap_branch_len**, codap_branch_len_context)		vlclbf
ntriangles -= 2		
}		
else		
ntriangles--		
}		
}		

5.9.2.2.1.15 triangle_data

triangle_data(i) {	No. of bits	Mnemonic
qf_decode(**triangulated**, triangulated_context)		vlclbf
depth=0		
root_triangle()		
for (i=1; i<ntriangles; i++)		
triangle(i)		
}		

5.9.2.2.1.16 root_triangle

root_triangle() {	No. of bits	Mnemonic
if (marching_triangle)		
qf_decode(**marching_pattern**, marching_pattern_context[**marching_pattern**])		vlclbf
else {		
if (3D_MOBL_start_code == "partition_type_2")		
if (tt_leaf == '0' && depth==0)		
qf_decode(**td_orientation**, td_orientation_context)		vlclbf
if (tt_leaf == '0')		
depth++		
else		
depth--		
}		
if (3D_MOBL_start_code == "partition_type_2")		
if (triangulated == '0')		
qf_decode(**polygon_edge**, polygon_edge_context[**polygon_edge**])		vlclbf
root_coord()		
root_normal()		
root_color()		
root_texCoord()		
}		

root_coord() {	No. of bits	Mnemonic
if (3D_MOBL_start_code == "partition_type_2") {		
if (visited[vertex_index] == 0) {		
root_coord_sample()		
if (visited[vertex_index] == 0) {		
coord_sample()		
coord_sample()		
}		
}		
}		
else {		
root_coord_sample()		
coord_sample()		

coord_sample()		
}		
}		

root_normal() {	No. of bits	Mnemonic
if (normal_binding != “not_bound”)		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (normal_binding != “bound_per_vertex” ‖ visited[vertex_index] == 0) {		
root_normal_sample()		
if (normal_binding != “bound_per_face” && (normal_binding != “bound_per_vertex” ‖ visited[vertex_index] == 0)) {		
normal_sample()		
normal_sample()		
}		
}		
else {		
root_normal_sample()		
if (normal_binding != “bound_per_face”) {		
normal_sample()		
normal_sample()		
}		
}		
}		

root_color() {	No. of bits	Mnemonic
if (color_binding != “not_bound”)		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (color_binding != “bound_per_vertex” ‖ visited[vertex_index] == 0) {		
root_color_sample()		
if (color_binding != “bound_per_face” && (color_binding != “bound_per_vertex” ‖ visited[vertex_index] == 0)) {		
color_sample()		
color_sample()		
}		
}		
else {		
root_color_sample()		
if (color_binding != “bound_per_face”) {		
color_sample()		
color_sample()		
}		
}		
}		

root_texCoord() {	No. of bits	Mnemonic
if (texCoord_binding != “not_bound”)		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (texCoord_binding != “bound_per_vertex” || visited[vertex_index] == 0) {		
root_texCoord_sample()		
if (texCoord_binding!= “bound_per_vertex” || visited[vertex_index] == 0) {		
texCoord_sample()		
texCoord_sample()		
}		
}		
else {		
root_texCoord_sample()		
texCoord_sample()		
texCoord_sample()		
}		
}		

5.9.2.2.1.17 triangle

triangle(i) {	No. of bits	Mnemonic
if (marching_triangle)		
qf_decode(**marching_edge**, marching_edge_context[**marching_edge**])		vlclbf
else {		
if (3D_MOBL_start_code == “partition_type_2”)		
if (tt_leaf == ‘0’ && depth==0)		
qf_decode(**td_orientation**, td_orientation_context)		vlclbf
if (tt_leaf == ‘0’)		
depth++		
else		
depth--		
if (triangulated == ‘0’)		
qf_decode(**polygon_edge**, polygon_edge_context[**polygon_edge**])		vlclbf
coord()		
normal()		
color()		
texCoord()		
}		

coord() {	No. of bits	Mnemonic
if (3D_MOBL_start_code == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_coord_sample()		
else		

coord_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
coord_sample()		
}		
}		

normal() {	No. of bits	Mnemonic
if (normal_binding == “bound_per_vertex”) {		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_normal_sample()		
else		
normal_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
normal_sample()		
}		
} else if (normal_binding == “bound_per_face”) {		
if (triangulated == ‘1’ || polygon_edge == ‘1’)		
normal_sample()		
} else if (normal_binding == “bound_per_corner”) {		
if (triangulated == ‘1’ || polygon_edge == ‘1’) {		
normal_sample()		
normal_sample()		
}		
normal_sample()		
}		
}		

color() {	No. of bits	Mnemonic
if (color_binding == “bound_per_vertex”) {		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_color_sample()		
else		
color_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
color_sample()		
}		
} else if (color_binding == “bound_per_face”) {		
if (triangulated == ‘1’ || polygon_edge == ‘1’)		

color_sample()		
} else if (color_binding == “bound_per_corner”) {		
if (triangulated == ‘1’ || polygon_edge == ‘1’) {		
color_sample()		
color_sample()		
}		
color_sample()		
}		
}		

texCoord() {	No. of bits	Mnemonic
if (texCoord_binding == “bound_per_vertex”) {		
if (3D_MOBL_start_code == “partition_type_2”) {		
if (visited[vertex_index] == 0)		
if (no_ancestors)		
root_texCoord_sample()		
else		
texCoord_sample()		
}		
else {		
if (visited[vertex_index] == 0)		
texCoord_sample()		
}		
} else if (texCoord_binding == “bound_per_corner”) {		
if (triangulated == ‘1’ || polygon_edge == ‘1’) {		
texCoord_sample()		
texCoord_sample()		
}		
texCoord_sample()		
}		
}		

root_coord_sample() {	No. of bits	Mnemonic
for (i=0; i<3; i++)		
for (j=0; j<coord_quant; j++)		
qf_decode(**coord_bit**, zero_context)		vlclbf
}		

root_normal_sample() {	No. of bits	Mnemonic
for (i=0; i<1; i++)		
for (j=0; j<normal_quant; j++)		
qf_decode(**normal_bit**, zero_context)		vlclbf
}		

root_color_sample() {	No. of bits	Mnemonic
for (i=0; i<3; i++)		
for (j=0; j<color_quant; j++)		
qf_decode(**color_bit**, zero_context)		vlclbf
}		

root_texCoord_sample() {	No. of bits	Mnemonic
for (i=0; i<2; i++)		
for (j=0; j<texCoord_quant; j++)		
qf_decode(**texCoord_bit**, zero_context)		vlclbf
}		

coord_sample() {	No. of bits	Mnemonic
for (i=0; i<3; i++) {		
j=0		
do {		
qf_decode(**coord_leading_bit**, coord_leading_bit_context[3*j+i])		vlclbf
j++		
} while (j<coord_quant && coord_leading_bit == '0')		
if (coord_leading_bit == '1') {		
qf_decode(**coord_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**coord_trailing_bit**, zero_context)		vlclbf
} while (j<coord_quant)		
}		
}		
}		

normal_sample() {	No. of bits	Mnemonic
for (i=0; i<1; i++) {		
j=0		
do {		
qf_decode(**normal_leading_bit**, normal_leading_bit_context[j])		vlclbf
j++		
} while (j<normal_quant && normal_leading_bit == '0')		
if (normal_leading_bit == '1') {		
qf_decode(**normal_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**normal_trailing_bit**, zero_context)		vlclbf
} while (j<normal_quant)		
}		
}		
}		

color_sample() {	No. of bits	Mnemonic
for (i=0; i<3; i++) {		
j=0		
do {		
qf_decode(**color_leading_bit**, color_leading_bit_context[3*j+i])		vlclbf
j++		
} while (j<color_quant && color_leading_bit == '0')		
if (color_leading_bit == '1') {		
qf_decode(**color_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**color_trailing_bit**, zero_context)		vlclbf
} while (j<color_quant)		
}		
}		
}		

texCoord_sample() {	No. of bits	Mnemonic
for (i=0; i<2; i++) {		
j=0		
do {		
qf_decode(**texCoord_leading_bit**, texCoord_leading_bit_context[2*j+i])		vlclbf
j++		
} while (j<texCoord_quant && texCoord_leading_bit == '0')		
if (texCoord_leading_bit == '1') {		
qf_decode(**texCoord_sign_bit**, zero_context)		vlclbf
do {		
qf_decode(**texCoord_trailing_bit**, zero_context)		vlclbf
} while (j<texCoord_quant)		
}		
}		
}		

5.9.2.2.1.18 stitching

stitching() {	No. of bits	Mnemonic
has_vertex_increase	1	bslbf
has_face_increase	1	bslbf
if (has_vertex_increase) {		
n_vertex_stitches	bitsPerV	uimsbf
for(int i = 0; i < n_vertex_stitches; i++){		
n_duplication_per_vertex_stitches	bitsPerV	uimsbf
for(int j = 0; j < n_duplication_per_vertex_stitches; j++) {		
vertex_index	bitsPerV	uimsbf
}		
}		
}		

if (has_face_increase){		
n_face_stitches	bitsPerF	uimsbf
for(int i = 0; i < n_face_stitches; i++){		
n_duplication_per_face_stitches	bitsPerF	uimsbf
for(int j = 0; j < n_duplication_per_face_stitches; j++){		
face_index	bitsPerF	uimsbf
}		
}		
}		
}		

5.9.2.2.1.19 3D_Mesh_Object_ Extension _Layer

3D_Mesh_Object_Extension_Layer() {	**No. of bits**	**Mnemonic**
if(vertex_order_flag) {		
if(vertex_order_per_CC_flag)		
vertex_order_per_CC_header()		
vertex_order()		
}		
if(face_order_flag) {		
if(face_order_per_CC_flag)		
face_order_per_CC_header()		
face_order()		
}		
}		

5.9.2.2.1.20 vertex_order_per_CC_header

vertex_order_per_CC_header () {	**No. of bits**	**Mnemonic**
for(i=0;i<nCC;i++) {		
nVOffset	16	uimsbf
for (j=0;j<nVOffset;j++)		
{		
vo_offset	24	uimsbf
firstVID	24	uimsbf
}		
}		
}		

5.9.2.2.1.21 face_order_per_CC_header

face_order_per_CC_header () {	No. of bits	Mnemonic
for(i=0;i<nCC;i++) {		
nFOffset	16	uimsbf
for (j=0;j<nFOffset;j++)		
{		
fo_offset	24	uimsbf
firstFID	24	uimsbf
}		
}		
}		

5.9.2.2.1.22 vertex_order

vertex_order() {	No. of bits	Mnemonic
for(i=0;i<nCC;i++) {		
for(bpvi=init_bpvi; bpvi >0; bpvi --)		
for(j=DecodingVertices;j>0;j--)		
vo_decode(**vo_id**,bpvi)	bpvi	uimsbf
}		
}		

5.9.2.2.1.23 face_order

face_order () {	No. of bits	Mnemonic
for(i=0;i<nCC;i++) {		
for(bpfi =init_bpfi; bpfi >0; bpfi --)		
for(j=DecodingFaces;j>0;j--)		
fo_decode(**fo_id,** bpfi)	bpfi	uimsbf
}		
}		

5.9.2.3 Visual bitstream semantics

5.9.2.3.1 3D Mesh Object

5.9.2.3.1.1 3D_Mesh_Object

3D_MO_start_code: This is a unique 16-bit code that is used for synchronization purpose. The value of this code is always '0000 0000 0010 0000'.

5.9.2.3.1.2 3D_Mesh_Object_Header

ccw: This boolean value indicates if the vertex ordering of the decoded faces follows a counter clock-wise order.

convex: This boolean value indicates if the model is convex.

solid: This boolean value indicates if the model is solid.

creaseAngle: This 6-bit unsigned integer indicates the crease angle.

3DMC_extension: This boolean value indicates if one or more of the 3DMC extension functionalities (vertex order and face order preserving and efficient texture mapping) are used.

5.9.2.3.1.3 3D_Mesh_Object_Layer

3D_MOL_start_code: This is a unique 16-bit code that is used for synchronization purposes. The value of this code is always '0000 0000 0011 0000'.

mol_id: This 8-bit unsigned integer specifies a unique id for the mesh object layer. Value 0 indicates a base layer. The first 3D_Mesh_Object_Layer immediately after a 3D_Mesh_Object_Header must have mold_id=0, and subsequent 3D_Mesh_Object_Layer's within the same 3D_Mesh_Object must have mold_id>0.

5.9.2.3.1.4 3D_Mesh_Object_Base_Layer

3D_MOBL_start_code: This is a code of length 16 that is used for synchronization purposes. It also indicates three different partition types for error resilience.

Table AMD1-1 — Definition of partition type information

3D_MOBL_start_code	partition type	Meaning
'0000 0000 0011 0001'	partition_type_0	One or more groups of vg, tt and td.
'0000 0000 0011 0011'	partition_type_1	One or more vgs
'0000 0000 0011 0100'	partition_type_2	One pair of tt and td.

mobl_id: This 8-bit unsigned integer specifies a unique id for the mesh object component.

one_bit: This boolean value is always true. This value is used for byte alignment.

last_component: This boolean value indicates if there are more connected components to be decoded. If **last_component** is '1', then the last component has been decoded. Otherwise there are more components to be decoded. This field is arithmetic coded

codap_last_vg – This boolean value indicates if the current vg is the last one in the partition. The value is false if there are more vgs to be decoded in the partition.

codap_vg_id: This unsigned integer indicates the id of the vertex graph corresponding to the current simple polygon in partition_type_2. The length of this value is a log scaled value of the vg_number of vg decoded from the previous partition_type_1. If there is only one vg in the previous partition_type_1,

codap_left_bloop_idx: This unsigned integer indicates the left starting index, within the bounding loop table of a connected component, for the triangles that are to be reconstructed in a partition. The length of this value is the log scaled value of the size of the bounding loop table.

codap_right_bloop_idx: This unsigned integer indicates the right starting index, within the bounding loop table of a connected component, for the triangles that are to be reconstructed in a partition. The length of this value is the log scaled value of the size of the bounding loop table.

codap_bdry_pred: This boolean value denotes how to predict geometry and photometry information that are in common with two or more partitions. If **codap_bdry_pred** is '1', the restricted boundary prediction mode is used, otherwise, the extended boundary prediction mode is used.

5.9.2.3.1.5 coord_header

coord_binding: This 2 bit unsigned integer indicates the binding of vertex coordinates to the 3D mesh. Table AMD1-2 shows the admissible values for **coord_binding**.

Table AMD1-2 — Admissible values for coord_binding

coord_binding	binding
00	forbidden
01	bound_per_vertex
10	forbidden
11	forbidden

coord_bbox: This boolean value indicates whether a bounding box is provided for the geometry. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **coord_xmin**=0, **coord_ymin**=0, **coord_zmin**=0, and **coord_size**=1.

coord_xmin, **coord_ymin**, **coord_zmin**: These floating point values indicate the lower left corner of the bounding box in which the geometry lies.

coord_size: This floating point value indicates the size of the bounding box.

coord_quant: This 5-bit unsigned integer indicates the quantisation step used for geometry. The minimum value of coord_quant is 1 and the maximum is 24.

coord_pred_type: This 2-bit unsigned integer indicates the type of prediction used to reconstruct the vertex coordinates of the mesh. Table AMD1-3 shows the admissible values for coord_pred_type.

Table AMD1-3 — Admissible values for coord_pred_type

coord_pred_type	prediction type
00	no_prediction
01	forbidden
10	parallelogram_prediction
11	reserved

coord_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict geometry. The only admissible value of **coord_nlambda** is 3. Table AMD1-4 shows the admissible values as a function of **coord_pred_type.**

Table AMD1-4 — Admissible values for coord_nlambda as a function of coord_prediction type

coord_pred_type	coord_nlambda
00	not coded
10	3

coord_lambda: This signed fixed-point number indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **coord_quant** + 3. The 3 leading bits represent the integer part, and the **coord_quant** remaining bits the fractional part.

5.9.2.3.1.6 normal_header

normal_binding: This 2 bit unsigned integer indicates the binding of normals to the 3D mesh. The admissible values are described in Table AMD1-5.

Table AMD1-5 — Admissible values for normal_binding

normal_binding	binding
00	not_bound
01	bound_per_vertex
10	bound_per_face
11	bound_per_corner

normal_bbox: This boolean value should always be false ('0').

normal_quant: This 5-bit unsigned integer indicates the quantisation step used for normals. The minimum value of normal_quant is 3 and the maximum is 31.

normal_pred_type: This 2-bit unsigned integer indicates how normal values are predicted. Table AMD1-6 shows the admissible values, and Table AMD1-7 shows admissible values as a function of normal_binding.

Table AMD1-6 — Admissible values for normal_pred_type

normal_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

Table AMD1-7 — Admissible combinations of normal_binding and normal_pred_type

normal_binding	normal_pred_type
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_face	no_prediction, tree_prediction
bound_per_corner	no_prediction, tree_prediction

normal_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **normal_nlambda** are 1, 2, and 3. Table AMD1-8 shows admissible values as a function of **normal_pred_type.**

Table AMD1-8 — Admissible values for normal_nlambda as a function of normal_prediction type

normal_pred_type	normal_nlambda
no_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

normal_lambda: This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for normal_lambda is (normal_quant-3)/2+3. The 3 leading bits represent the integer part, and the **normal_quant** remaining bits the fractional part.

5.9.2.3.1.7 color_header

color_binding: This 2 bit unsigned integer indicates the binding of colors to the 3D mesh. Table AMD1-9 shows the admissible values.

Table AMD1-9 — Admissible values for color_binding

color_binding	Binding
00	not_bound
01	bound_per_vertex
10	bound_per_face
11	bound_per_corner

color_bbox: This boolean indicates if a bounding box for colors is given. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **color_rmin**=0, **color_gmin**=0, **color_bmin**=0, and **color_size**=1.

color_rmin, **color_gmin**, **color_bmin**: These floating point values give the position of the lower left corner of the bounding box in RGB space.

color_size: This floating point value gives the size of the color bounding box.

color_quant: This 5-bit unsigned integer indicates the quantisation step used for colors. The minimum value of color_quant is 1 and the maximum is 16.

color_pred_type: This 2-bit unsigned integer indicates how colors are predicted. Table AMD1-10 shows the admissible values, and Table AMD1-11 shows admissible values as a function of color_binding.

Table AMD1-10 — Admissible values for color_pred_type

color_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

Table AMD1-11 — Admissible combinations of color_binding and color_pred_type

color_binding	color_pred_type
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_face	no_prediction, tree_prediction
bound_per_corner	no_prediction, tree_prediction

color_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **color_nlambda** are 1, 2, and 3. Table AMD1-12 shows admissible values as a function of **normal_pred_type.**

Table AMD1-12 — Admissible values for color_nlambda as a function of color_prediction type

color_pred_type	color_nlambda
no_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

color_lambda: This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **color_quant** + 3. The 3 leading bits represent the integer part, and the **normal_quant** remaining bits the fractional part.

5.9.2.3.1.8 texCoord_header

texCoord_binding: This 2 bit unsigned integer indicates the binding of texture coordinates to the 3D mesh. Table AMD1-13describes the admissible values.

Table AMD1-13 — Admissible values for texCoord_binding

texCoord_binding	Binding
00	not_bound
01	bound_per_vertex
10	forbidden
11	bound_per_corner

texCoord_bbox: This boolean value indicates if a bounding box for texture coordinates is given. If no bounding box is provided, a default bounding box is used. The default bounding box is defined as **texCoord_umin**=0, **texCoord_vmin**=0, and **texCoord_size**=1.

texCoord_umin, **texCoord_vmin**: These floating point values give the position of the lower left corner of the bounding box in 2D space.

texCoord_size: This floating point value gives the size of the texture coordinate bounding box.

texCoord_quant: This 5-bit unsigned integer indicates the quantisation step used for texture coordinates. The minimum value of texCoord_quant is 1 and the maximum is 16.

texCoord_pred_type: This 2-bit unsigned integer indicates how colors are predicted. Table AMD1-14 shows the admissible values, and Table AMD1-15 shows admissible values as a function of texCoord_binding.

Table AMD1-14 — Admissible values for texCoord_pred_type

texCoord_pred_type	prediction type
00	no_prediction
01	forbidden
10	parallelogram_prediction
11	reserved

Table AMD1-15 — Admissible combinations of texCoord_binding and texCoord_pred_type

texCoord_binding	texCoord_pred_type
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_corner	no_prediction, tree_prediction

texCoord_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **texCoord_nlambda** are 1, 2, and 3. Table AMD1-16 shows admissible values as a function of **texCoord_pred_type.**

Table AMD1-16 — Admissible values for texCoord_nlambda as a function of texCoord_prediction type

texCoord_pred_type	texCoord_nlambda
not_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

texCoord_lambda: This signed fixed-point indicates the weight given to an ancestor for prediction. The number of bits used for this field is equal to **texCoord_quant** + 3. The 3 leading bits represent the integer part, and the **texCoord_quant** remaining bits the fractional part.

5.9.2.3.1.9 3DMC_extension_header

function_type: This 4-bit unsigned integer indicates the function type supported in 3DMC extension. Table AMD1-17 shows the admissible values for **function_type.**

Table AMD1-17 — Admissible values for function type

function_type_code	function_type
0000	Order_mode
0001	Adaptive_quant_texCoord_mode
0010~1110	Reserved
1111	Escape code

5.9.2.3.1.10 Order_mode_header

vertex_order_flag: This boolean value indicates whether the vertex order is provided or not.

vertex_order_per_CC_flag: This boolean value indicates whether the vertex orders are coded at the unit of connected component or not.

face_order_flag: This boolean value indicates whether the face order is provided or not.

face_order_per_CC_flag: This boolean value indicates whether the face orders are coded at the unit of connected component or not.

5.9.2.3.1.11 Adaptive_quant_texCoord_mode_header

texCoord_quant_u: This 16-bit unsigned integer indicates the value of quantisation step size for direction u(x).

texCoord_quant_v: This 16-bit unsigned integer indicates the value of quantisation step size for direction v(y).

5.9.2.3.1.12 connected component

has_stitches: This boolean value indicates if stitches are applied for the current connected component (within itself or between the current component and connected components previously decoded) This field is arithmetic coded.

5.9.2.3.1.13 vertex_graph

vg_simple: This boolean value indicates if the current vertex graph is simple. A simple vertex graph does not contain any loop. This field is arithmetic coded.

vg_last: This boolean value indicates if the current run is the last run starting from the current branching vertex. This field is not coded for the first run of each branching vertex, i.e. when the skip_last variable is true. When not coded the value of **vg_last** for the current vertex run is considered to be false. This field is arithmetic coded.

vg_forward_run: This boolean value indicates if the current run is a new run. If it is not a new run, it is a previously traversed run, indicating a loop in the graph . This field is arithmetic coded.

vg_loop_index: This unsigned integer indicates the index of run to which the current loop connects. Its unary representation (see Table AMD1-18) is arithmetic coded. If the variable openloops is equal to **vg_loop_index**, the trailing '1' in the unary representation is omitted.

Table AMD1-18 — Unary representation of the vg_loop_index field

vg_loop_index	unary representation
0	1
1	01
2	001
3	0001
4	00001
5	000001
6	0000001
...	
openloops-1	openloops-1 0's

vg_run_length: This unsigned integer indicates the length of the current vertex run. Its unary representation (see Table AMD1-19) is arithmetic coded.

Table AMD1-19 — Unary representation of the vg_run_length field

vg_run_length	unary representation
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
N	n-1 0's followed by 1

vg_leaf: This boolean value indicates if the last vertex of the current run is a leaf vertex. If it is not a leaf vertex, it is a branching vertex. This field is arithmetic coded.

vg_loop: This boolean value indicates if the leaf of the current run connects to a branching vertex of the graph, indicating a loop. This field is arithmetic coded.

5.9.2.3.1.14 triangle_tree

branch_position: This integer variable is used to store the last branching triangle in a partition.

tt_run_length: This unsigned integer indicates the length of the current triangle run. Its unary representation (see Table AMD1-20) is arithmetic coded.

Table AMD1-20 — Unary representation of the tt_run_length field

tt_run_length	unary representation
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
N	n-1 0's followed by 1

tt_leaf: This boolean value indicates if the last triangle of the current run is a leaf triangle. If it is not a leaf triangle, it is a branching triangle. This field is arithmetic coded.

triangulated: This boolean value indicates if the current component contains triangles only. This field is arithmetic coded.

marching_triangle: This boolean value is determined by the position of the triangle in the triangle tree. If **marching_triangle** is 0, the triangle is a leaf or a branch. Otherwise, the triangle is a run.

marching_edge: This boolean value indicates the marching edge of an edge inside a triangle run. If **marching_edge** is false, it stands for a march to the left, otherwise it stands for a march to the right. This field is arithmetic coded.

polygon_edge: This boolean value indicates whether the base of the current triangle is an edge that should be kept when reconstructing the 3D mesh object. If the base of the current triangle is not kept, the edge is discarded. This field is arithmetic coded.

codap_branch_len: This unsigned integer indicates the length of the next branch to be traversed. The length of this value is the log scaled value of the size of the bounding loop table.

5.9.2.3.1.15 triangle

td_orientation: This boolean value informs the decoder the traversal order of tt/td pair at a branch. This field is arithmetic coded. Table AMD1-21 shows the admissible values.

Table AMD1-21 — Admissible values for td_orientation

td_orientation	traversal order
0	right branch first
1	left branch first

visited: This variable indicates if the current vertex has been visited or not. When **codap_bdry_pred** is '1', visited is true for the vertices visited in the current partition. However, when **codap_bdry_pred** is '0', visited is true for the vertices visited in the previous partitions as well as in the current partition.

vertex_index: This variable indicates the index of the current vertex in the vertex array.

no_ancestors: This boolean value is true if there are no ancestors to use for prediction of the current vertex.

coord_bit: This boolean value indicates the value of a geometry bit. This field is arithmetic coded.

coord_leading_bit: This boolean value indicates the value of a leading geometry bit. This field is arithmetic coded.

coord_sign_bit: This boolean value indicates the sign of a geometry sample. This field is arithmetic coded.

coord_trailing_bit: This boolean value indicates the value of a trailing geometry bit. This field is arithmetic coded.

normal_bit: This boolean value indicates the value of a normal bit. This field is arithmetic coded.

normal_leading_bit: This boolean value indicates the value of a leading normal bit. This field is arithmetic coded.

normal_sign_bit: This boolean value indicates the sign of a normal sample. This field is arithmetic coded.

normal_trailing_bit: This boolean value indicates the value of a trailing normal bit. This field is arithmetic coded.

color_bit: This boolean value indicates the value of a color bit. This field is arithmetic coded.

color_leading_bit: This boolean value indicates the value of a leading color bit. This field is arithmetic coded.

color_sign_bit: This boolean value indicates the sign of a color sample. This field is arithmetic coded.

color_trailing_bit: This boolean value indicates the value of a trailing color bit. This field is arithmetic coded.

texCoord_bit: This boolean value indicates the value of a texture bit. This field is arithmetic coded.

texCoord_leading_bit: This boolean value indicates the value of a leading texture bit. This field is arithmetic coded.

texCoord_sign_bit: This boolean value indicates the sign of a texture sample. This field is arithmetic coded.

texCoord_trailing_bit: This boolean value indicates the value of a trailing texture bit. This field is arithmetic coded.

5.9.2.3.1.16 stitching

has_vertex_increase: This boolean value indicates if the vertex increase is occurred during the conversion of non-orientable/non-manifold model into an oriented-manifold model and stitching information.

has_face_increase: This boolean value indicates if the face increase is occurred during the conversion of non-orientable/non-manifold model into an oriented-manifold model and stitching information.

n_vertex_stitches: This *bitsPerV*-bit unsigned integer specifies how many vertex stitching operations are needed to reconstruct original non-orientable/non-manifold model. The value of *bitsPerV* is set to $\lceil \log_2 nV \rceil$, where nV means the total number of vertices.

n_duplication_per_vertex_stitches: This *bitsPerV*-bit unsigned integer specifies the number of duplication in vertices for each vertex stitch operation. The value of *bitsPerV* is set to $\lceil \log_2 nV \rceil$, where nV means the total number of vertices.

vertex_index: This *bitsPerV*-bit unsigned integer specifies a unique index of the vertex order. The value of *bitsPerV* is set to $\lceil \log_2 nV \rceil$, where nV means the total number of vertices.

n_face_stitches: This *bitsPerF*-bit unsigned integer specifies how many face stitching operations are needed to reconstruct original non-orientable/non-manifold model. The value of *bitsPerF* is set to $\lceil \log_2 nF \rceil$, where nF means the total number of faces.

n_duplication_per_face_stitches: This *bitsPerF*-bit unsigned integer specifies the number of duplication in faces for each face stitch operation. The value of *bitsPerF* is set to $\lceil \log_2 nF \rceil$, where nF means the total number of faces.

face_index: This *bitsPerF*-bit unsigned integer specifies a unique index of the face order. The value of *bitsPerF* is set to $\lceil \log_2 nF \rceil$, where nF means the total number of faces.

5.9.2.3.1.17 vertex_order_per_CC_heaer

nVOffset: This 16-bit unsigned integer indicates the number of offset values within the given connected component.

vo_offset: This 24-bit unsigned integer indicates the offset value needed to reconstruct the vertex order at the unit of IndexedFaceSet.

firstVID: This 24-bit unsigned integer indicates the first vertex index within the given connected component using **vo_offset** as its offset value.

5.9.2.3.1.18 face_order_per_CC_heaer

nFOffset: This 16-bit unsigned integer indicates the number of offset values within the given connected component.

fo_offset: This 24-bit unsigned integer indicates the offset value needed to reconstruct the face order at the unit of IndexedFaceSet.

firstFID: This 24-bit unsigned integer indicates the first face index within the given connected component using **fo_offset** as its offset value.

5.9.2.3.1.19 vertex_order

vo_id: This bpvi-bit unsigned integer specifies a unique index of the vertex order based on the input IndexedFaceSet. The value of *init_bpvi* is set to $\lceil \log_2 nV \rceil$, where nV is the total number of vertices. The value of *bpvi* and *DecodingVertices* are explicitely described in subclause 5.9.3.1.8.

5.9.2.3.1.20 face_order

fo_id: This bpfi-bit unsigned integer specifies a unique index of the face order based on the input IndexedFaceSet. The value of *init_bpfi* is set to $\lceil \log_2 nF \rceil$, where nF is the total number of faces. The value of *bpfi* and *DecodingFaces* are explicitely described in subclause 5.9.3.1.9.

5.9.3 The visual decoding process

5.9.3.1 3D Mesh Object Decoding

The Topological Surgery decoder is composed of eight main modules, as shown in Figure AMD1-6, namely:

— An arithmetic decoder, which reads a section of the input stream and outputs a bit stream.

— A vertex graph decoder, which reads a section of the bit stream and outputs a bounding loop look-up table.

— A triangle tree decoder, which reads a section of the bit stream and outputs the length of each run and the size of each sub-tree of the triangle tree.

— A stitch decoder, which reads a section of the bit stream and outputs stitching information.

— A vertex order decoder, which reads a section of bit stream and outputs a sequence of vertex orders.

— A face order decoder, which reads a section of bit stream and outputs a sequence of face orders.

— An order fixer operator, which takes the output data produced by the vertex graph decoder, the triangle tree decoder, the triangle data decoder, the vertex order decoder, and the face order decoder, and applies the rearrange operation according to the decoded vertex and face order information.

— A triangle data decoder, which reads a section of the input bit and outputs a stream of triangle data. This stream of triangle data contains the geometry and the properties associated with each triangle.

When a 3D mesh bitstream is received, the control information including partition types and the header information including properties are obtained through the demultiplexer. The remaining bitstream is fed into the arithmetic decoder.

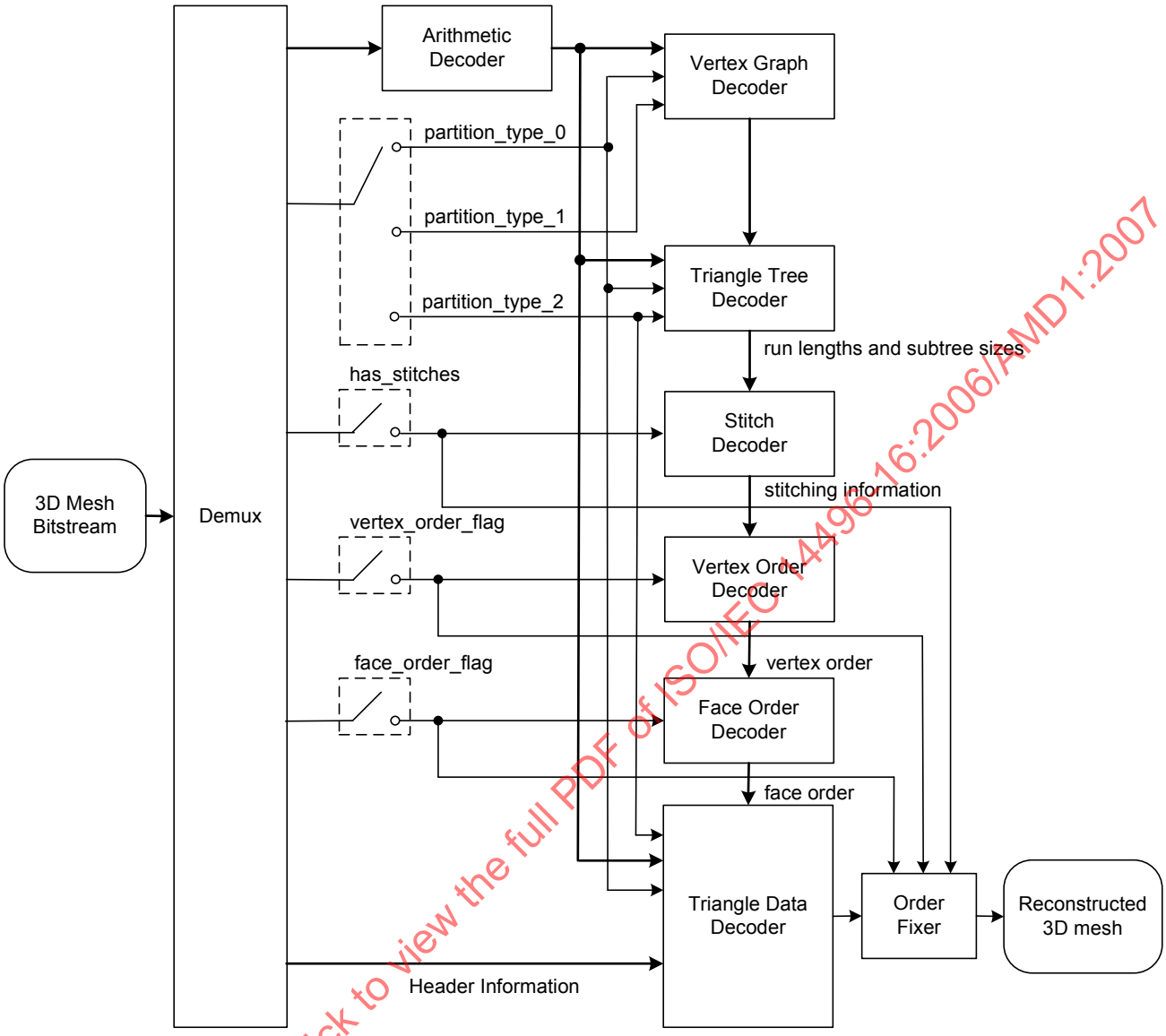

Figure AMD1-6 — Block diagram of the Topological Surgery decoder

5.9.3.1.1 Start codes and bit stuffing

A start code is a two-byte code of the form '0000 0000 00xx xxxx'. Several such codes are inserted into the bitstream for synchronisation purposes. To prevent any wrongful synchronisation, such codes shall not be emulated by the data. This is guaranteed by the insertion of a stuffing bit '1' after each byte-aligned sequence of eight 0's. The decoder shall skip these stuffing bits when parsing the bit stream. Note that the arithmetic coder is designed such as to never generate a synchronisation code. Therefore the bit skipping rule need not be applied to the portions of the bit stream that contain arithmetic coded data.

5.9.3.1.2 The Topological Surgery decoding process.

The connectivity of a 3D mesh is represented as a Simple Polygon (triangulated with a single boundary loop) with zero or more pairs of boundary vertices identified, and with zero or more internal edges labelled as polygon edges. When the pairs of corresponding boundary edges are identified, the edges of the resulting reconstructed mesh corresponding to boundary edges of the Simple Polygon form the Vertex Graph. The

correspondence among pairs of Simple Polygon boundary edges is recovered by the decoding process from the structure of Vertex Graph, producing the Vertex Loop look-up table. The Vertex Graph is represented as (i) a rooted spanning tree, (ii) the Vertex Tree, and (iii) zero or more jump edges. The Simple Polygon is represented as (i) a rooted spanning tree, (ii) the Triangle Tree, which defines an order of traversal of the triangles, (iii) a sequence of marching patterns, and (iv) a sequence of polygon_edges. The marching patterns are used to reconstruct the triangles by marching on the left or on the right along the polygon bounding loop, starting from an initial edge called the root edge. The polygon edges are used to join or not to join triangles sharing a marching edge to form the faces of the mesh. As the triangles of the simple polygon are visited in the order of traversal of the Triangle Tree, simple polygon boundary edges are put in correspondence using the information contained in the Vertex Loop look-up table, and so, reconstructing the original connectivity.

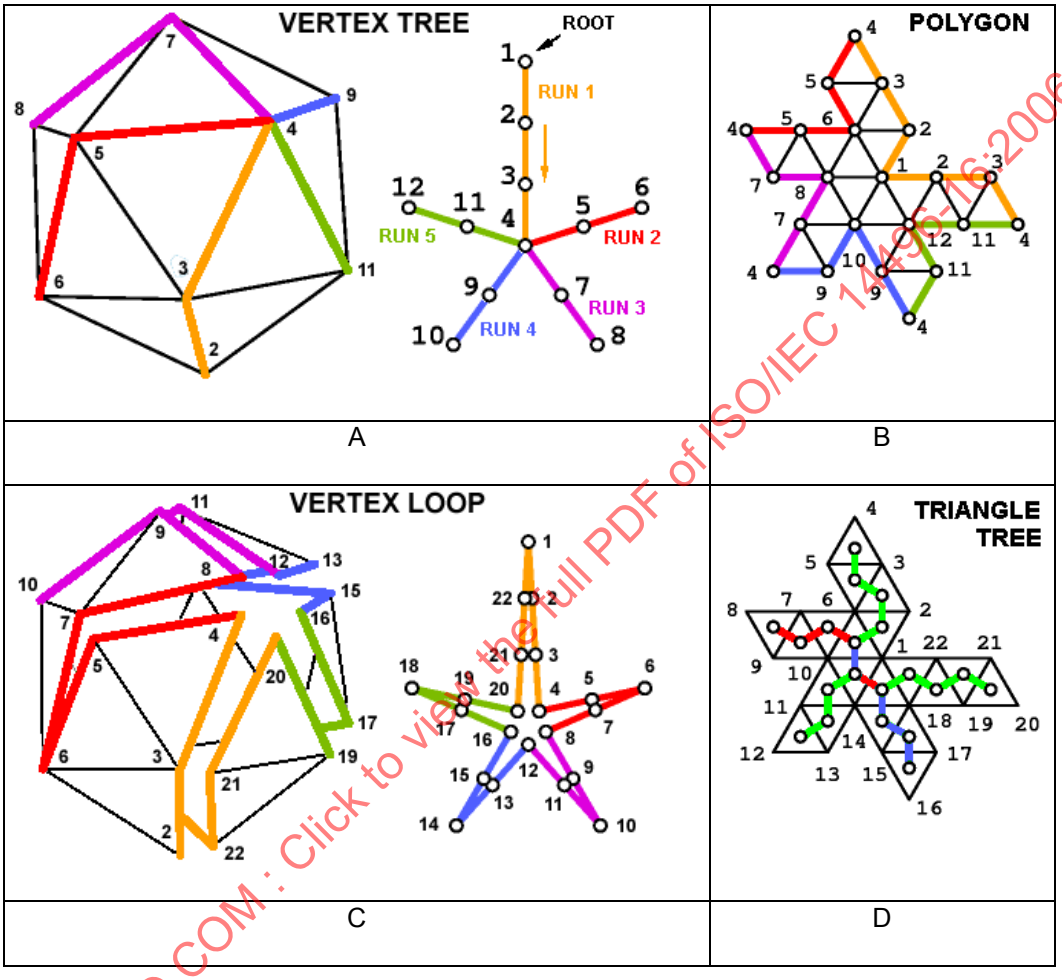

Figure AMD1-7 — Topological Surgery representation of a simple mesh.
A : for a simple mesh, the vertex graph is a tree.
B : when a simple mesh is cut through the vertex tree, the result is a simple polygon.
C : each edge of the vertex tree corresponds to exactly two boundary edges of the simple polygon.
D : the triangle tree spanning the dual graph of the simple polygon. In the general case, the face forest is first constructed spanning the dual graph of the 3D mesh. The vertex graph is composed of the remaining edges of the 3D mesh. If the 3D mesh has polygonal faces, they are subsequently triangulated by inserting virtual marching edges, converting the face forest into a triangle forest with one triangle tree per connected component.

The geometry and property data are quantised and predicted as a function of ancestors in the order of traversal of the triangles. The corresponding prediction errors are transmitted in the same order of traversal, interlaced with the corresponding marching and polygon_edges, so that, as soon as all these triangle data are received, the decoder can output the corresponding triangle.

At a high level, the Topological Surgery decoder performs the following steps:

```
Decode header information
for each partition {
 if (partition_type is 0) {
    for each connected component {
       decode vertex graph and construct bounding loop table
       decode triangle tree and construct table of triangle tree run lengths
       for each triangle in simple polygon {
          decode marching field and polygon field, and reconstruct
connectivity of triangle
          decode and reconstruct vertex coordinates and properties
       }
    }
 }
 else if (partition_type is 1) {
    for each vertex graph {
       decode vertex graph and construct bounding loop table
    }
 }
 else if (partition_type is 2) {
    decode partition header information
    decode triangle tree and construct table of triangle tree run lengths
    for each triangle in simple polygon {
       decode marching field, orientation field, and polygon field, and
reconstruct connectivity of triangle
       decode and reconstruct vertex coordinates and properties
    }
 }
}
```

5.9.3.1.3 Header decoder

The header information is divided into three parts. The first part contains information about the high level shading properties of the model, and the second defines the properties that are bound to the mesh. The third part contains information about the extension properties.

5.9.3.1.3.1 High level shading properties

These are the **ccw**, **solid**, **convex**, **creaseAngle** fields defined in an IndexedFaceSet node. The crease angle is quantised to a 6-bit value and is reconstructed as 2*pi***creaseAngle**/63.

5.9.3.1.3.2 Property bindings and quantiser scales

There are four kinds of properties: vertex coordinates (coord), normals (normal), colors (color) and texture coordinates (texCoord). Properties can be bound to the mesh in four different ways: no binding, per vertex, per face and per corner. Not all combinations of bindings are valid. Table AMD1-22 lists the valid combinations. The binding of each property is obtained from coord_binding, normal_binding, color_binding and texCoord_binding, respectively.

Table AMD1-22 — List of valid combinations of properties and bindings

	no binding	per vertex	per face	per corner
coord	forbidden	valid	forbidden	forbidden
normal	valid	valid	valid	valid
color	valid	valid	valid	valid
texCoord	valid	valid	forbidden	valid

For each property for which there is a binding, i.e. the binding field does not contain the no binding value, the following information is further decoded: a bounding box, a quantisation step, a prediction mode, a list of coefficients used for linear prediction. The decoding process for the vertex coordinates is further described below. The same decoding process applies to the other properties.

The presence of a bounding box is given by the field **coord_bbox**. The bounding box is a cube represented by the position of its lower left corner and the length of its side. For geometry these parameters are given by **coord_xmin**, **coord_ymin**, **coord_zmin** and **coord_size**. If no bounding box is coded, as default bounding box is assumed. The default bounding box has its lower left corner at the origin, and a unit size.

The next field **coord_quant** indicates the number of bits to which each coordinate is quantised to. The **coord_pred_type** field indicates the prediction mode. It should always be equal to '10'.

The **coord_nlambda** field indicates the number of coefficients used for linear prediction. It can take one of three values, namely '01', '10', and '11'. **coord_nlambda**-1 number of values are then read from the **coord_lambda** field. The last coord_lambda value is computed so that the sum of all the coord_nlamda value is equal to one. This field indicates the weight given to an ancestor for prediction. The floating point value of a coefficient is given by the decoded signed integer divided by 2 to the power **coord_quant**. The last coefficient is never transmitted and is defined to be 1 minus the sum of all other coefficients.

There are some restrictions for the normal property: the **normal_bbox** field is always false, and the **normal_quant** value must always an odd number.

5.9.3.1.3.3 Extension Properties

5.9.3.1.3.3.1.1 Order mode

There is vertex order and face order preserving mode. The presence of the vertex orders is given by the field **vertex_order flag** and the next field **vertex_order_per_CC_flag** field indicates whether the vertex orders are coded at the unit of IndexedFaceSet or connected component.

The presence of the face orders is given by the field **face_order flag** and the next field **face_order_per_CC_flag** field indicates whether the face orders are coded at the unit of IndexedFaceSet or connected component.

If the vertex orders are encoded at the unit of connected component, there are **nVOffset**, **vo_offset**, and **firstVID** fields defined in vertex_order_per_CC_header. The **nVOffset** field indicates the number of offset values within the given connected component, The **vo_offset** field indicates the offset value and the next field **firstVID** indicates the first vertex index within the given connected component using **vo_offset** as its offset value.

If the face orders are encoded at the unit of connected component, there are **nFOffset**, **fo_offset**, and **firstFID** fields defined in face_order_per_CC_header. The **nFOffset** field indicates the number of offset values within the given connected component, The **fo_offset** field indicates the offset value and the next field **firstFID** indicates the first face index within the given connected component using **fo_offset** as its offset value.

The decoding process for the vertex and face orders is further described below.

5.9.3.1.3.3.1.2 Adaptive quantisation for texCoord mode

These are the **texCoord_quant_u** and **tex_Coord_quant_v** fields defined in Adaptive_quant_texCoord_mode_header. **texCoord_quant_u** indicates the value of quantisation step size for direction u(x) and **texCoord_quant_v** indicates the value of quantisation step size for direction v(y).

5.9.3.1.4 Partition type

There are three partition types to convey vertex graph(vg), triangle tree(tt), and triangle data(td). The partition type specifies admissible combinations of these three pieces of data in the bitstream. See Figure AMD1-8.

sc	vg	tt	td	last_component	...	vg	tt	td	last_component

(a)

sc	vg	last_vg	...	vg	last_vg

(b)

sc	codap_vg_id	left/codap_right_bloop_idx	codap_bdry_pred	tt	codap_branch_len	td

(c)

Figure AMD1-8 — Types of data partition : (a) one or more group of vg, tt and td included in a partition. (b) one or more vgs included in a partition. (c) the partition has one pair of tt and td.

5.9.3.1.4.1 connected component partition (partition_type_0)

This partition indicates the bitstream with one or more sequence of connected component consisting of vertex graph, triangle tree, and triangle data. The end of the connected component sequence is determined by the **last_component** field. The following partition may be either type 0 or 1.

5.9.3.1.4.2 vertex graph partition (partition_type_1)

This partition indicates the bitstream with one or more sequence of vertex graph. The end of the vertex graph sequence is determined by the **last_vg** field. The partition following the vertex graph partition shall be tt/td pair partition.

5.9.3.1.4.3 tt/td pair partition (partition_type_2)

This partition indicates the bitstream with one pair of triangle tree and triangle data. The tt/td pair partition is characterized by the vertex graph id, visiting indices, and boundary prediction mode. These variables are given fields **codap_vg_id, left/codap_right_bloop_idx**, and **codap_bdry_pred. codap_vg_id** field indicates a vertex graph corresponding to the tt/td pair. It is used to get the bounding loop information from the vertex graph. **codap_left_bloop_idx** and **codap_right_bloop_idx** fields indicate the left and right starting points of the bounding loop. They are also used to decide if a vertex in the partition is already visited in the previous partition. **codap_bdry_pred** field indicates if the vertices on the boundary of the partition should be decoded when the vertex is shared with previous partitions. If the partition ends at a branch, **codap_branch_len** field is added to the bitstream. Any type of partition may follow the tt/td pair partition.

5.9.3.1.4.4 Restricted boundary prediction mode (codap_bdry_pred=0)

The restricted boundary prediction mode does not duplicate vertices between partitions. Since the vertices predicted in the previous partitions may not be available, prediction is done only with the available vertices which are predicted in the current partition. If the previous partitions are lost, the triangles at the boundaries may not be reconstructed due to the fact that the vertices from the previous partitions are not available. However, the vertices predicted in the current partition may be reconstructed and decoding can continue.

The following process is applied for the prediction with a subset of all ancestors. Let the current method for prediction with 3 ancestors (a,b,c) be $d' = f(a,b,c)$. Then the equation is $d = d' + e$ and the value e is encoded. However, when the ancestors are visited in the previous partitions, the prediction method is as follows.

if (all ancestors (a, b, and c) are not available) then $d' = 0$

else if (only one ancestor, say t, is available) then $d' = t$

else if (two ancestors, say t_1 and t_2, are available) then

if (both ancestors are edge distance 1 from current vertex) then $d' = (t_1 + t_2)/2$

else if (t_1 is edge distance 1 from current vertex) then $d' = t_1$

else $d' = t_2$

else $d' = f(a,b,c)$.

5.9.3.1.4.5 Extended boundary prediction mode (codap_bdry_pred=1)

When this mode is selected, the decoder assumes that every vertex is not visited in the previous partition. Thus, after the three vertices of the root triangle are predicted, the vertices in the rest of the triangles in the partition will have all three ancestors for prediction.

5.9.3.1.5 Vertex Graph Decoder

The vertex graph decoder reconstructs a sequence of vertex runs. It then constructs the Vertex Loop look-up table.

For meshes that have simple topology and that do not have a boundary, the vertex graph does not contain any loops and is hence a tree. The field **vg_simple,** if set, indicates that the graph contains no loop. Some fields below are skipped when **vg_simple** is set.

Each vertex run is characterized by several variables, namely length, last, forward, loop and loop index. These variables are given by the fields **vg_run_length**, **vg_last**, **vg_forward_run**, **vg_loop** and **loop_index**. **vg_forward_run**, **vg_loop** and **loop_index** contain information relative to loops. When **vg_simple** is true, these three fields are not coded.

The process to reconstruct the vertex loop uses an auxiliary loop queue variable, initially empty. When **vg_loop** is set the current run is put into the loop queue. When **vg_forward_run** is not set a run is pulled out from the queue. **loop_index** determines the position of the pulled out run in the queue. **vg_forward_run** is not coded when the loop queue is empty. **loop_index** is coded by its unary representation. When the index is equal to the number of elements in the queue minus 1, the trailing bit of its unary representation is not coded. Therefore when there is a single element in the queue, **loop_index** is not coded.

The end of the graph is determined by the **vg_last** and **vg_leaf** fields. An auxiliary depth variable is used for this purpose. It is decremented each time **vg_last** is set and incremented each time **vg_leaf** is not set. The depth is initialised to zero at the beginning of the graph. The end of the graph is reached when the depth variable becomes negative.

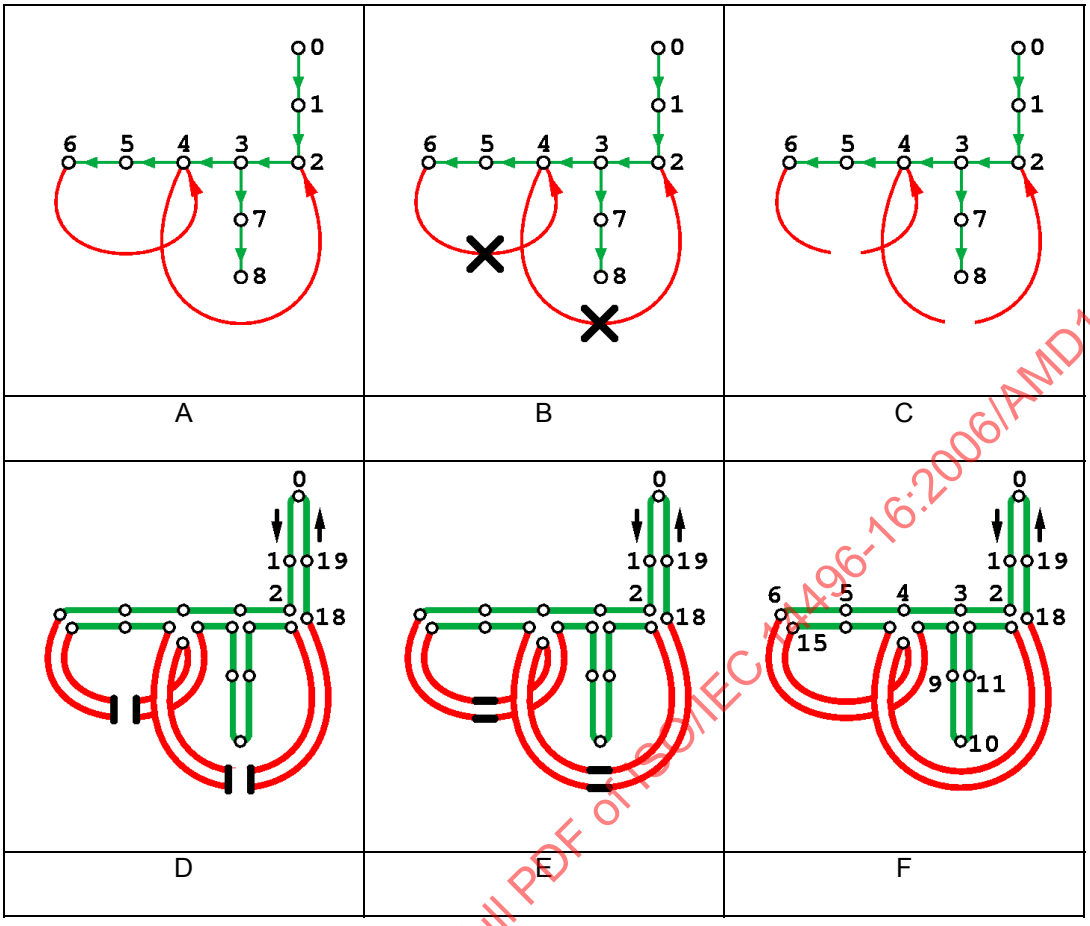

Figure AMD1-9 — Steps to build the bounding loop from the vertex graph.
A : vertex graph decomposed into vertex tree (green) and jump edges (red).
B : extended vertex tree is created by cutting jump edges in half.
C : the extended vertex tree has two leaves for each jump edge.
D : build the extended vertex tree loop.
E : connect the start and end of each jump edge.
F : resulting boundary loop.

Vertex numbers are assigned according to a depth-first traversal of the graph. The bounding loop look-up table is constructed by going around the graph and for each traversed vertex recording its number.

5.9.3.1.6 Triangle Tree Decoder

The triangle tree decoder reconstructs a sequence of triangle runs. For each run it generates a run length and the size of its sub-tree.

The triangle tree is a sequence of triangle runs. For each run a length **tt_run_length** and a boolean flag **tt_leaf** is given. A branching run is a run for which **tt_leaf** is false and a leaf run a run for which **tt_leaf** is true. The decoder stops decoding the sequence of runs when the number of decoded leaf runs is superior to the number of branching runs. In the syntax a variable depth is used that counts the number of branching runs minus the number of leaf runs. This implies that the total number of runs is always an odd number.

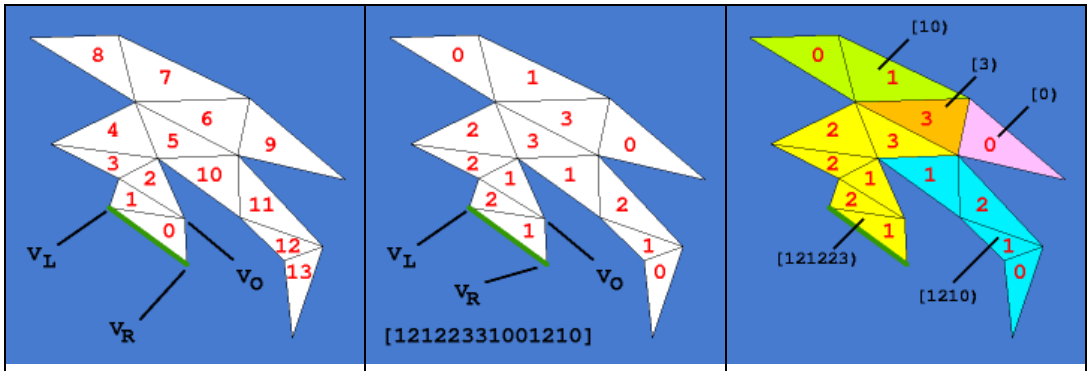

Figure AMD1-10 — Representing a simple polygon as a table of triangle runs.
A: depth-first order of traversal of the triangle tree starting from the root edge (green).
B: each triangle is classified by a 2-bit code as leaf(0), advance-right (1), advance-left (2), or branching (3).
C : The sequence of 2-bit codes is partitioned into runs ending in branching or leaf triangles. Each run is encoded as a (tt_run_length,tt_leaf) pair, plus a sequence of marching_edge bits.
Optionally, in the case of 3D meshes with polygonal faces, an additional sequence of polygon_edge bits is used to indicate which internal edges of the simple polygon correspond to original internal edges of the mesh, and which ones to virtual internal edges.

The **tt_leaf** flag is also used for the computation of the length of each run.

If the length of each run is correctly decoded and does not have virtual triangles, then the sum of the length of all runs must be equal to the length of the bounding loop constructed by the vertex graph decoder minus 2 (assuming that the bounding loop is correctly reconstructed).

However, if a run is partitioned, the triangle tree bitstream contains one or two virtual leaf triangles. Hence when decoding triangle data, the triangle data corresponding to the virtual leaf triangles shall not be decoded.

There are one or two virtual triangles if the number of triangles in the current partition is less than **codap_right_bloop_idx** - **codap_left_bloop_idx** - 1. Otherwise there are no virtual triangles. If there are virtual triangles and the third-last triangle is a branching triangle, the last two triangles in the partition are the virtual triangles. Other wise only the last triangle is virtual.

5.9.3.1.7 Stitch Decoder

There is an increase in the number of vertices or/and faces after conversion of non-orientable/non-manifold model into an oriented-manifold model and stitching information as described in Figure AMD1-11.

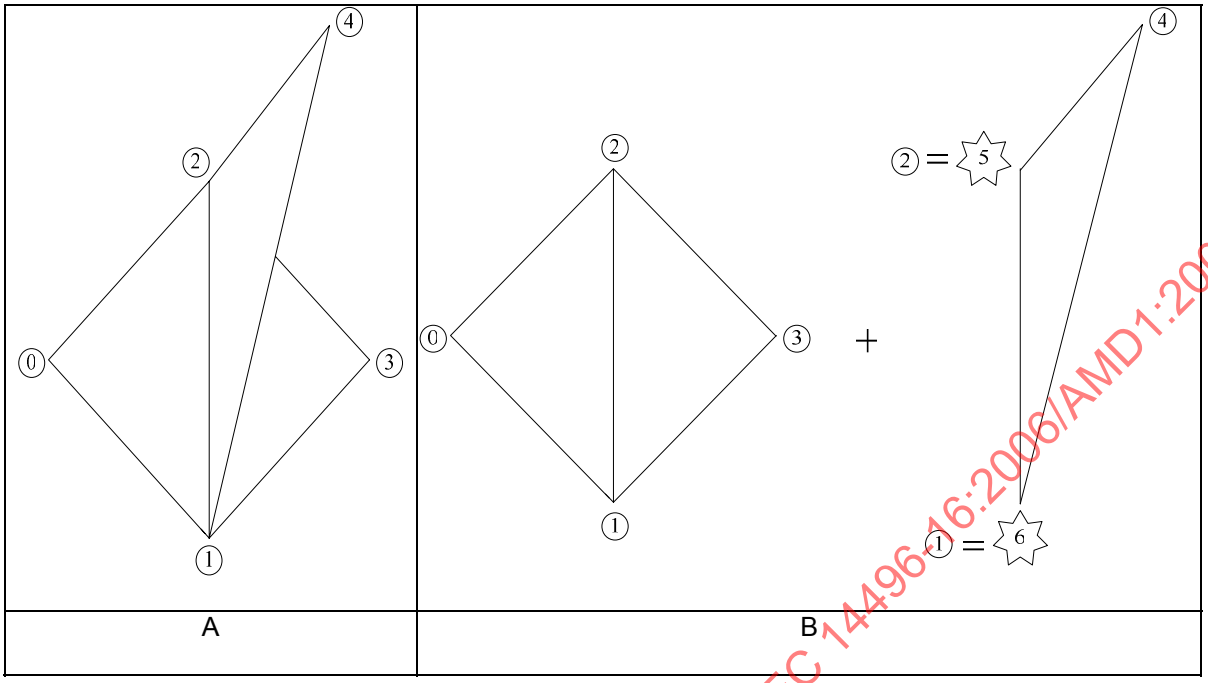

Figure AMD1-11 — A non-manifold 3D mesh (A) and converted manifold meshes (B)

The stitch decoder reconstructs the stitching information to reconstruct the original non-orientable/non-manifold model. The following pseudo-code summarizes the operation of the Stitch decoder: if the boolean **has_stitches** flag is true, then Stitch decoder decodes stitching information from the bitstream.

If the boolean value **has_vertex_increase** is true, then vertex increase related stitching information is decoded; An unsigned integer **n_vertex_stitches** is decoded associated with the number of vertex stitching information; For each vertex stitching information, the unsigned integer **n_duplication_per_vertex_stitches** is decoded describing the number of duplications, and multiple unsigned integer **vertex_index** are decoded associated with the actual index of the original vertex and duplicated vertex index.

If the boolean value **has_face_increase** is true, then face increase related stitching information is decoded; An unsigned integer **n_face_stitches** is decoded associated with the number of face stitching information; For each face stitching information, the unsigned integer **n_duplication_per_face_stitches** is decoded describing the number of duplications, and multiple unsigned integer **face_index** are decoded associated with the actual index of the original face and duplicated face index.

```
// reading stitch information from the bitstream
 int bitsPerV = representBit(totalVertices);
 int bitsPerF = representBit(totalFaces);

if (ibstrm.getBit(has_vertex_increase)== false) return false;
if (ibstrm.getBit(has_face_increase)== false) return false;

if (has_vertex_increase) {
        ibstrm.getMBitInt(bitsPerV, n_vertex_stitches);
        for(int i = 0; i < n_vertex_stitches; i++){
                ibstrm.getMBitInt(bitsPerV, n_duplication_per_vertex_stitches);
```

```
            for(int j = 0; j < n_duplication_per_vertex_stitches; j++){
                ibstrm.getMBitInt(bitsPerV, vertex_index);
                vertexStitch.push_back(vertex_index);
            }
            vertexStitch.push_back(-1);
        }
}

if (has_face_increase) {
        ibstrm.getMBitInt(bitsPerF, n_face_stitches);
        for(int i = 0; i < n_face_stitches; i++){
            ibstrm.getMBitInt(bitsPerF, n_duplication_per_face_stitches);
            for(int j = 0; j < n_duplication_per_face_stitches; j++){
                ibstrm.getMBitInt(bitsPerF, face_index);
                faceStitch.push_back(face_index);
            }
            faceStitch.push_back(-1);
        }
}
```

Table AMD1-23 lists the encoded data for the given non-manifold model described in Figure AMD1-12.

Table AMD1-23 — Example of the encoded data

Fields	Encoded data (integer)	Encoded data (bits)
has_vertex_increase	1	1
has_face_increase	0	0
n_vertex_stitches	2 (‘2=5’ and ‘1=6’)	010 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
n_duplication_per_vertex_stitches	2 (‘2=5’)	010 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
vertex_index	2 (original vertex index)	010 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
vertex_index	5 (copied vertex index)	101 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
n_duplication_per_vertex_stitches	2 (‘1=6’)	010 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
vertex_index	1 (original vertex index)	001 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)
vertex_index	6 (copied vertex index)	110 (bitsPerV=3 = $\lceil \log_2 7 \rceil$)

Using decoded stitching information, the original non-manifold/non-orientable model can be reconstructed by deleting the copied vertices and replacing the incidence of copied vertices with the incidence of original vertex.

5.9.3.1.8 Vertex Order Decoder

The vertex order decoder reconstructs a sequence of vertex order, **vo_id**, based on the input IndexedFaceSet. Note that this sequence is empty if there is no vertex order information. If this sequence is not empty, the decoder decodes the sequence of vertex order information which is contained in the 3D_Mesh_Object_Extension_Layer. The header for the presence of vertex order, which is a boolean flag **vertex_order_flag**, is defined in the Order_mode_header.

```
// vertex order decoder

int nV = number of vertices of the current connected component;

// compute the number of bits(bpvi) required to decode the vertex order in the initial state
init_bpvi = bpvi=bitsToRepresent(nV);

// vertexVector: for containing not decoded vertex indices
vector <int> vertexVector;

// initialize vertexVector to all vertex indices
for(int q = 0; q < nV; q++)
   vertexVector.push_back(q);

// for bpvi from init_bpvi to 1 : last vertex order is not transmitting
for(i=init_bpvi;i>0;i--) {
   // compute the distinguishable unit for decoding current bpvi bits uniquely
   DecodingVertices = nV- pow(2, (bpvi-1));

   // delList : for containing decoded vertex indices
   int* delList = new int[DecodingVertices];

   // compute remaining vertices after decoding the current distinguishable unit
   nV=nV-DecodingVertices;
   for(j=nV;j>0;j--) {
     vo_decode(vo_id, bpvi);
   }

   // sort the delList
   qsort(delList, delCnt, sizeof(int), compare);

   // update vertexVector by deleting decoded vo_id
   for(q = 0; q < delCnt; q++)
```

```
        vertexVector.erase(vertexVector.begin() + delList[q] - q);
}

// last vertex order is calculated by the remaining one in the vertexVector
qVertexIndex.push_back(vertexVector[0]);
```

Figure AMD1-12 — vertex order decoder pseudo-code

5.9.3.1.8.1 Decoding the symbol of the vertex order

After reading the bpvi bits from the bit stream, the vertex order is calculated as follows: If bpvi is same as init_bpvi, bpvi is transformed into unsigned integer value and this unsigned integer value becomes the vertex order. If bpvi is not same as init_bpvi, bpvi is transformed into unsigned integer value but this unsigned integer value is not the vertex order. The vertex order is the (unsigned inter value)-th value among the remaining consecutive vertex indices in the range 0, …, (V-1).

```
void vo_decode(unsigned int vo_id, bpvi){
    ibstrm.getMOBitInt(bpvi, VertexIndex);
    if (bpvi==init_bpvi){
        vo_id=vertexIndex
    }
    else {
        // vertexVector[VertexIndex] is the (VertexIndex)-th vo_id that does not decoded
        vo_id = vertexVector[VertexIndex];
    }
    qVertexIndex.push_back(vo_id);
    //vo_id add to delList for updating vertexVector
    delList[delCnt++] = vo_id;
}
```

If the vertex order is coded at the unit of connected component, the calculated vertex order using the aforementioned function should be transformed into the final vertex order represented at the unit of IndexedFaceSet. Otherwise, if the vertex order is coded at the unit of IndexedFaceSet, the calculated vertex order will be the final vertex order.