INTERNATIONAL STANDARD

ISO/IEC 14496-16

Third edition
2009-12-15

Information technology — Coding of audio-visual objects —

Part 16: Animation Framework eXtension (AFX)

Technologies de l'information — Codage des objets audiovisuels — Partie 16: Extension du cadre d'animation (AFX)

Reference number
ISO/IEC 14496-16:2009(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

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.

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

This third edition cancels and replaces the second edition (ISO/IEC 14496-16:2006), of which it constitutes a minor revision. It also incorporates ISO/IEC 14496-16:2006/Amd.1:2007, ISO/IEC 14496-16:2006/Amd.1:2007/Cor.1:2008, ISO/IEC 14496-16:2006/Amd.1:2007/Cor.2:2008, ISO/IEC 14496-16:2006/Amd.2:2009, ISO/IEC 14496-16:2006/Amd.3:2008.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects [Technical Report]*

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description*

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

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

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

— *Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)*

— *Part 21: MPEG-J Graphics Framework eXtensions (GFX)*

— *Part 22: Open Font Format*

— *Part 23: Symbolic Music Representation*

— *Part 24: Audio and systems interaction [Technical Report]*

— *Part 25: 3D Graphics Compression Model*

— *Part 27: 3D Graphics conformance*

Information technology — Coding of audio-visual objects —

Part 16:
Animation Framework eXtension (AFX)

1 Scope

This Part of ISO/IEC 14496 specifies MPEG-4 Animation Framework eXtension (AFX) model for representing and encoding 3D graphics assets to be used standalone or integrated in interactive multimedia presentations (the latter when combined with other parts of MPEG-4). Within this model, MPEG-4 is extended with higher-level synthetic objects for geometry, texture, and animation as well as dedicated compressed representations.

AFX also specifies a backchannel for progressive streaming of view-dependent information.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14496-1:2004, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-2:2004, *Information technology — Coding of audio-visual objects — Part 2: Visual*

ISO/IEC 14496-11:2005, *Information technology — Coding of audio-visual objects — Part 11: Scene description and application engine*

ISO/IEC 14496-25:2008, *Information technology — Coding of audio-visual objects — Part 25: 3D Graphics Compression Model*

3 Symbols and abbreviated terms

List of symbols and abbreviated terms.

AFX	Animation Framework eXtension
BIFS	BInary Format for Scene
DIBR	Depth-Image Based Representation
ES	Elementary Stream
IBR	Image-Based Rendering
NDT	Node Data Type

OD Object Descriptor

VRML Virtual Reality Modelling Language

Animation Framework eXtension – 3D Graphics Primitives

3.1 Introduction

MPEG-4, ISO/IEC 14496, is a multimedia standard that enables composition of multiple audio-visual objects on a terminal. Audio and visual objects can come from *natural* sources (e.g. a microphone, a camera) or from *synthetic* ones (i.e. made by a computer); each source is called a *media* or a *stream*. On their terminals, users can display, play, and interact with MPEG-4 audio-visual contents, which can be downloaded previously or streamed from remote servers. Moreover, each object may be protected to ensure a user has the right credentials before downloading and displaying it.

Unlike natural audio and video objects, computer graphics objects are purely synthetic. Mixing computer graphics objects with traditional audio and video enables augmented reality applications, i.e. applications mixing natural and synthetic objects. Examples of such contents range from DVD menus, and TV's Electronic Programming Guides to medical and training applications, games, and so on.

Like other computer graphics specifications, MPEG-4 synthetic objects are organized in a *scene graph* based on VRML97 [1], which is a direct acyclic tree where *nodes* represent objects and branches their properties, called *fields*. As each object can receive and emit events, two branches can be connected by the means of a *route*, which propagates events from one field of one node to another field of another node. As any other MPEG-4 media, scenes may receive updates from a server that modify the topology of the scene graph.

The Animation Framework eXtension (AFX)'s main objective is to propose compression scheme for static and animated 3D assets. This encoded version is encapsulated in an Elementary Stream, a similar approach as the one used by audio or video in MPEG-4. The AFX compression is closely connected to the definition of the graphics primitives in the scene graph. For such reason, AFX second objective is to update the scene graph when necessary and define new BIFS nodes for graphics primitives. However, AFX compression can also operate on other scene graph formalisms (as indicated in ISO/IEC 14496-25 a.k.a. MPEG-4 Part 25).

The place of AFX within MPEG-4 terminal architecture is illustrated in Figure 1.

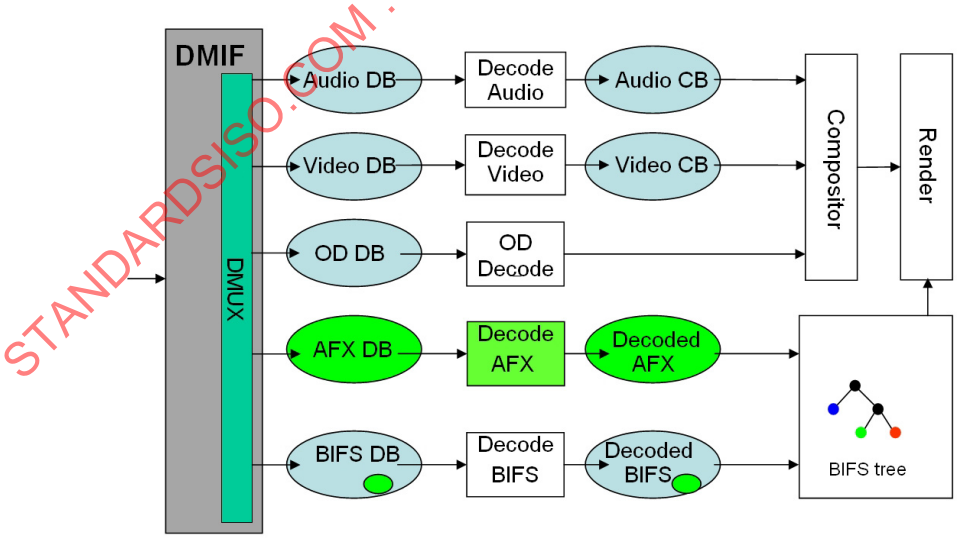


Figure 1 — Animation Framework eXtension (in green) and MPEG-4 Systems.

Figure 1 shows the position of the Animation Framework within MPEG-4 Terminal architecture. It extends the existing BIFS tree with new nodes and define the AFX streams that carry dedicated compressed object representations and a backchannel for view-dependent features.

3.2 The AFX place within computer animation framework

To understand the place of AFX within computer animation framework [35], let's take an example. Suppose one wants to build an avatar. The avatar consists of geometry elements that describe its legs, arms, head and so on. Simple geometric elements can be used and deformed to produce more physically realistic geometry. Then, skin, hair, cloths are added. These may be physic-based models attached to the geometry. Whenever the geometry is deformed, these models deform and thanks to their physics, they may produce wrinkles. Biomechanical models are used for motion, collision response, and so on. Finally, the avatar may exhibit special behaviors when it encounters objects in its world. It might also learn from experiences: for example, if it touches a hot surface and gets hurt, next time, it will avoid touching it. This hierarchy also works in a top to bottom manner: if it touches a hot surface, its behavior may be to retract its hand. Retracting its hand follows a biomechanical pattern. The speed of the movement is based on the physical property of its hand linked to the rest of its body, which in turn modify geometric properties that define the hand.

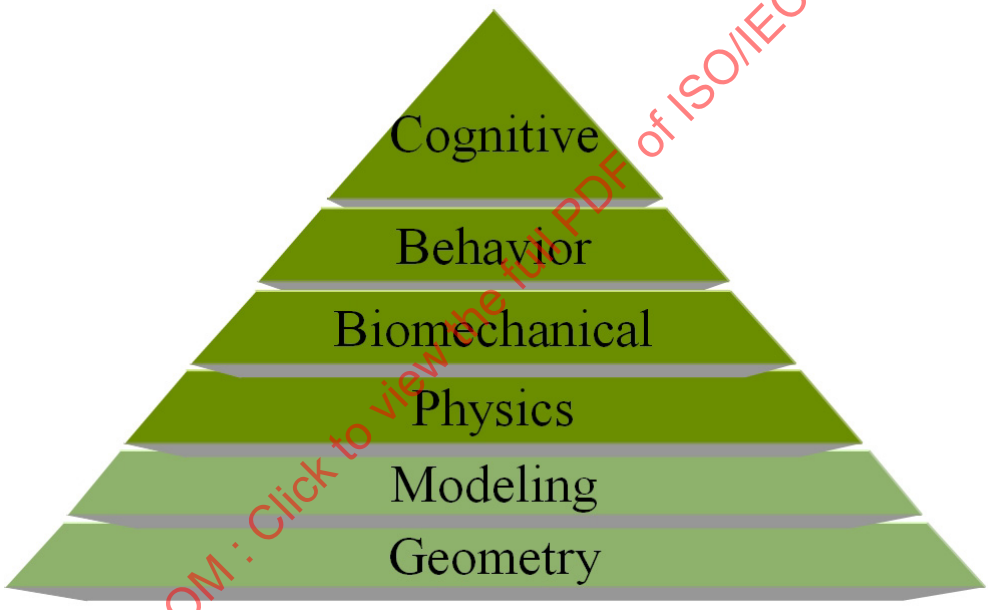


Figure 2 — Models in computer games and animation [35].

1) Geometric component. These models capture the form and appearance of an object. Many characters in animations and games can be quite efficiently controlled at this low-level. Due to the predictable nature of motion, building higher-level models for characters that are controlled at the geometric level is generally much simpler.

2) Modeling component. These models are an extension of geometric models and add linear and non-linear deformations to them. They capture transformation of the models without changing its original shape. Animations can be made on changing the deformation parameters independently of the geometric models.

3) Physical component. These models capture additional aspects of the world such as an object's mass inertia, and how it responds to forces such as gravity. The use of physical models allows many motions to be created automatically and with unparallel realism.

4) Biomechanical component. Real animals have muscles that they use to exert forces and torques on their own bodies. If we already have built physical models of characters, they can use virtual muscles to move themselves around. These models have their roots in control theory.

5) Behavioral component. A character may expose a reactive behavior when its behavior is solely based on its perception of the current situation (i.e. no memory of previous situations). Goal-directed behaviors can be used to define a cognitive character's goals. They can also be used to model flocking behaviors.

6) Cognitive component. If the character is able to learn from stimuli from the world, it may be able to adapt its behavior. These models are related to artificial intelligence techniques.

AFX specification currently deals with the first two categories. Models of the last two categories are typically application-specific and often designed programmatically. While the first four categories can be animated using existing tools such as interpolators, the last two categories have their own logic and cannot be animated the same way.

In each category, one can find many models for any applications: from simple models that require little processing power (low-level models) to more complex models (high-level models) that require more computations by the terminal. VRML [1] and BIFS specifications provide low-level models that belong to the Geometry and Modeling components.

Higher-level components can be defined as providing a compact representation of functionality in a more abstract manner. Typically, this abstraction leads to mathematical models that need few parameters. These models cannot be rendered directly by a graphic card: internally, they are converted to low-level primitives a graphic card can render.

Besides a more compact representation, this abstraction often provides other functionalities such as view-dependent subdivision, automatic level-of-details, smoother graphical representation, scalability across terminals, and progressive local refinements. For example, a subdivision surface can be subdivided based on the area viewed by the user. For animations, piecewise-linear interpolators require few computations but require lots of data in order to represent a curved path. Higher-level animation models represent animation using piecewise-spline interpolators with less values and provide more control over the animation path and timeline.

In the remaining of this document, this conceptual organization of tools is followed in the same spirit an author will create content: the geometry is first defined with or without solid modeling tools, then texture is added to it. Objects can be deformed and animated using modeling and animation tools. Behavioral and Cognitive models can be programmatically implemented using JavaScript or MPEG-J defined in ISO/IEC 14496-11.

NOTE: Some generic tools developed originally within the Animation Framework eXtension have been relocated in ISO/IEC 14496-11 along with other generic tools. This includes:

- Spline-based generic animation tools, called Animator nodes;
- Optimized interpolator compression tools;
- BitWrapper node that enables compressed representation of exisiting nodes;
- Procedural textures based on fractal plasma.

3.3 Geometry tools

3.3.1 Non-Uniform Rational B-Spline (NURBS)

3.3.1.1 Introduction

A Non-Uniform Rational B-Spline (NURBS) curve of degree $p > 0$ (and hence of order $p+1$) and control points $\{\mathbf{P}_i\}$ ($0 \le i \le n-1$; $n \ge p+1$) is mathematically defined as [34], [60], [85]:

$$\mathbf{C}(u) = \sum_{i=0}^{n-1} R_{i,p}(u)\mathbf{P}_i = \frac{\sum_{i=0}^{n-1} N_{i,p}(u) w_i \mathbf{P}_i}{\sum_{i=0}^{n-1} N_{i,p}(u) w_i}$$

The parameter $u \in [0, 1]$ allows to travel along the curve and $\{R_{i,p}\}$ are its rational basis functions. The latter can in turn be expressed in terms of some positive (and not all null) weights $\{w_i\}$, and $\{N_{i,p}\}$, the p^{th}-degree B-Spline basis functions defined on the possibly non-uniform, but always non-decreasing knot sequence/vector of length $m = n + p+1$: $U = \{u_0, u_1, \ldots, u_{m-1}\}$, where $0 \le u_i \le u_{i+1} \le 1 \ \forall\ 0 \le i \le m - 2$.

The B-Spline basis functions are defined recursively using the Cox de Boor formula:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1}; \\ 0 & \text{otherwise;} \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u).$$

If $u_i = u_{i+1} = \ldots = u_{i+k-1}$, it is said that u_i has multiplicity k. $\mathbf{C}(u)$ is infinitely differentiable inside knot spans, *i.e.*, for $u_i < u < u_{i+1}$, but only $p-k$ times differentiable at the parameter value corresponding to a knot of multiplicity k, so setting several consecutive knots to the same value u_j decreases the smoothness of the curve at u_j. In general, knots cannot have multiplicity greater than p, but the first and/or last knot of U can have multiplicity $p+1$, *i.e.*, $u_0 = \ldots = u_p = 0$ and/or $u_{m-p-1} = \ldots = u_{m-1} = 1$, which causes $\mathbf{C}$ to interpolate the corresponding endpoint(s) of the control polygon defined by $\{\mathbf{P}_i\}$, *i.e.*, $\mathbf{C}(0) = \mathbf{P}_0$ and/or $\mathbf{C}(1) = \mathbf{P}_{n-1}$. Therefore, a knot vector of the kind $U = \left\{ \underbrace{u_0 = 0, \ldots, 0}_{p+1}, u_{p+1}, \ldots, u_{m-p-2}, \underbrace{1, \ldots, 1 = u_{m-1}}_{p+1} \right\}$ causes the curve to be endpoint interpolating, *i.e.*, to interpolate both endpoints of its control polygon. Extreme knots, multiple or not, may enclose any non-decreasing subsequence of interior knots: $0 < u_i \le u_{i+1} < 1$. An endpoint interpolating curve with no interior knots, *i.e.*, one with U consisting of $p+1$ zeroes followed by $p+1$ ones, with no other values in between, is a p^{th}-degree Bézier curve: *e.g.*, a cubic Bézier curve can be described with four control points (of which the first and last will lie on the curve) and a knot vector $U = \{0, 0, 0, 0, 1, 1, 1, 1\}$.

It is possible to represent all types of curves with NURBS and, in particular, all conic curves (including parabolas, hyperbolas, ellipses, etc.) can be represented using rational functions, unlike when using merely polynomial functions.

Other interesting properties of NURBS curves are the following:

- Affine invariance: rotations, translations, scalings, and shears can be applied to the curve by applying them to $\{\mathbf{P}_i\}$.

- Convex hull property: the curve lies within the convex hull defined by $\{\mathbf{P}_i\}$. The control polygon defined by $\{\mathbf{P}_i\}$ represents a piecewise approximation to the curve. As a general rule, the lower the degree, the closer the NURBS curve follows its control polygon.

- Local control: if the control point $\mathbf{P}_i$ is moved or the weight $\mathbf{w}_i$ is changed, only the portion of the curve swept when $u_i < u < u_{i+p+1}$ is affected by the change.

- NURBS surfaces are defined as tensor products of two NURBS curves, of possibly different degrees and/or numbers of control points:

$$\mathbf{C}(u,v) = \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{l-1} N_{i,p}(u)N_{j,q}(v)w_{i,j}\mathbf{P}_{i,j}}{\sum_{i=0}^{n-1}\sum_{j=0}^{l-1} N_{i,p}(u)N_{j,q}(v)w_{i,j}}$$

The two independent parameters u, $v \in [0, 1]$ allow to travel across the surface. The B-Spline basis functions are defined as previously, and the resulting surface has the same interesting properties that NURBS curves have. Multiplicity of knots may be used to introduce sharp features (corners, creases, etc.) in an otherwise smooth surface, or to have it interpolate the perimeter of its control polyhedron.

3.3.1.2 NurbsCurve

3.3.1.2.1 Node interface

NurbsCurve { #%NDT=SFGeometryNode

eventIn	MFInt32	set_colorIndex		
exposedField	SFColorNode	color	NULL	
exposedField	MFVec4f	controlPoint	[ ]	
exposedField	SFInt32	tessellation	0	# [0, ∞)
field	MFInt32	colorIndex	[ ]	# [-1, ∞)
field	SFBool	colorPerVertex	TRUE	
field	MFFloat	knot	[ ]	
field	SFInt32	order	4	# [3, 34]

}

3.3.1.2.2 Functionality and semantics

The **NurbsCurve** node describes a 3D NURBS curve, which is displayed as a curved line, similarly to what is done with the **IndexedLineSet** primitive.

The **order** field defines the order of the NURBS curve, which is its degree plus one.

The **controlPoint** field defines a set of control points in a coordinate system where the weight is the last component. The number of control points must be greater than or equal to the order of the curve. All weight values must be greater than or equal to 0, and at least one weight must be strictly greater than 0. If the weight of a control point is increased above 1, that point is more closely approximated by the surface. However the surface is not changed if all weights are multiplied by a common factor.

The **knot** field defines the knot vector. The number of knots must be equal to the number of control points plus the order of the curve, and they must be ordered non-decreasingly. By setting consecutive knots to the same value, the degree of continuity of the curve at that parameter value is decreased. If *o* is the value of the field **order**, *o* consecutive knots with the same value at the beginning (resp. end) of the knot vector cause the curve to interpolate the first (resp. last) control point. Other than at its extremes, there may not be more than *o*–1 consecutive knots of equal value within the knot vector. If the length of the knot vector is 0, a default knot vector consisting of *o* 0's followed by *o* 1's, with no other values in between, will be used, and a Bézier curve of degree *o*–1 will be obtained. A closed curve may be specified by repeating the starting control point at the end and specifying a periodic knot vector.

The **tessellation** field gives hints to the curve tessellator as to the number of subdivision steps that must be used to approximate the curve with linear segments: for instance, if the value *t* of this field is greater than or equal to that of the **order** field, *t* can be interpreted as the absolute number of tesselation steps, whereas $t = 0$ lets the browser choose a suitable tessellation.

Fields **color**, **colorIndex**, **colorPerVertex**, and **set_colorIndex** have the same semantic as for **IndexedLineSet** applied to the control points.

3.3.1.3 NurbsCurve2D

3.3.1.3.1 Node interface

NurbsCurve2D { #%NDT=SFGeometryNode

eventIn	MFInt32	set_colorIndex		
exposedField	SFColorNode	color	NULL	
exposedField	MFVec3f	controlPoint	[ ]	
exposedField	SFInt32	tessellation	0	# [0, ∞)
field	MFInt32	colorIndex	[ ]	# [-1, ∞)
field	SFBool	colorPerVertex	TRUE	
field	MFFloat	knot	[ ]	# (-∞, ∞)
field	SFInt32	order	4	# [3, 34]

}

3.3.1.3.2 Functionality and semantics

The **NurbsCurve2D** is the 2D version of **NurbsCurve**; it follows the same semantic as **NurbsCurve** with 2D control points.

3.3.1.4 NurbsSurface

3.3.1.4.1 Node interface

NurbsSurface { #%NDT=SFGeometryNode

eventIn	MFInt32	set_colorIndex		
eventIn	MFInt32	set_texCoordIndex		
exposedField	SFColorNode	color	NULL	
exposedField	MFVec4f	controlPoint	[]	
exposedField	SFTextureCoordinateNode	texCoord	NULL	
exposedField	SFInt32	uTessellation	0	# [0, ∞)
exposedField	SFInt32	vTessellation	0	# [0 ,∞)
field	MFInt32	colorIndex	[] # [-1, ∞)	
field	SFBool	colorPerVertex	TRUE	
field	SFBool	solid	TRUE	
field	MFInt32	texCoordIndex	[]	# [-1 ,∞)
field	SFInt32	uDimension	4	# [3, 258]
field	MFFloat	uKnot	[]	# (-∞,∞)
field	SFInt32	uOrder	4	# [3, 34]
field	SFInt32	vDimension	4	# [3, 258]
field	MFFloat	vKnot	[]	# (-∞,∞)
field	SFInt32	vOrder	4	# [3, 34]

}

3.3.1.4.2 Functionality and semantics

The **NurbsSurface** Node describes a 3D NURBS surface. Similar 3D surface nodes are **ElevationGrid** and **IndexedFaceSet**. In particular, **NurbsSurface** extends the definition of **IndexedFaceSet**. If an implementation does not support **NurbsSurface**, it should still be able to display its control polygon as a set of (triangulated) quadrilaterals, which is the coarsest approximation of the NURBS surface.

The **uOrder** field defines the order of the surface in the *u* dimension, which is its degree in the *u* dimension plus one. Similarly, the **vOrder** field define the order of the surface in the *v* dimension.

The **uDimension** and **vDimension** fields define respectively the number of control points in the *u* and *v* dimensions, which must be greater than or equal to the respective orders of the curve.

The **controlPoint** field defines a set of control points in a coordinate system where the weight is the last component. These control points form an array of dimension **uDimension** x **vDimension**, in a similar way to the regular grid formed by the control points of an **ElevationGrid**, the difference being that, in the case of a **NurbsSurface**, the points need not be regularly spaced. The number of control points in each dimension must be greater than or equal to the corresponding order of the curve. Specifically, the three spatial coordinates (*x*, *y*, *z*) and weight (*w*) of control point $\mathbf{P}_{i,j}$ (NB: $0 \le i <$ **uDimension** $\ge$ **uOrder** and $0 \le j <$ **vDimension** $\ge$ **vOrder**) are obtained as follows:

```
P[i,j].x = controlPoint[i + j*uDimension].x
P[i,j].y = controlPoint[i + j*uDimension].y
P[i,j].z = controlPoint[i + j*uDimension].z
P[i,j].w = controlPoint[i + j*uDimension].w
```

All **weight** values must be greater than or equal to 0, and at least one weight must be strictly greater than 0. If the weight of a control point is increased above 1, that point is more closely approximated by the surface. However the surface is not changed if all weights are multiplied by a common factor.

The **uKnot** and **vKnot** fields define the knot vectors in the *u* and *v* dimensions respectively, and their semantics are analogous to that of the **knot** field of the **NurbsCurve** node.

The **uTessellation** and **vTessellation** fields give hints to the surface tessellator as to the number of subdivision steps that must be used to approximate the surface with (triangulated) quadrilaterals: for instance, if the value *t* of the **uTessellation** field is greater than or equal to that of the **uOrder** field, *t* can be interpreted as the absolute number of tesselation steps in the *u* dimension, whereas *t* equals zero lets the browser choose a suitable tessellation.

solid is a rendering hints that indicates if the surface is closed. If FALSE, two-side lighting is enabled. If TRUE, only one-side lighting is applied.

color, **colorIndex**, **colorPerVertex**, **set_colorIndex**, **texCoord**, **texCoordIndex**, and **set_textCoordIndex** have the same functionality and semantics as for **IndexedFaceSet** and they apply on the control points defined in **controlPoint**, just like these fields apply on **coord** points of an **IndexedFaceSet**.

3.3.2 Subdivision surfaces

3.3.2.1 Introduction

Subdivision is a recursive refinement process that splits the facets or vertices of a polygonal mesh (the initial “control hull”) to yield a smooth limit surface. The refined mesh obtained after each subdivision step is used as the control hull for the next step, so all successive (and hierarchically nested) meshes can be regarded as control hulls. The refinement of a mesh is performed both on its topology, as the vertex connectivity is made richer and richer, and on its geometry, as the new vertices are positioned in such a way that the angles formed by the new facets are smaller than those formed by the old facets.

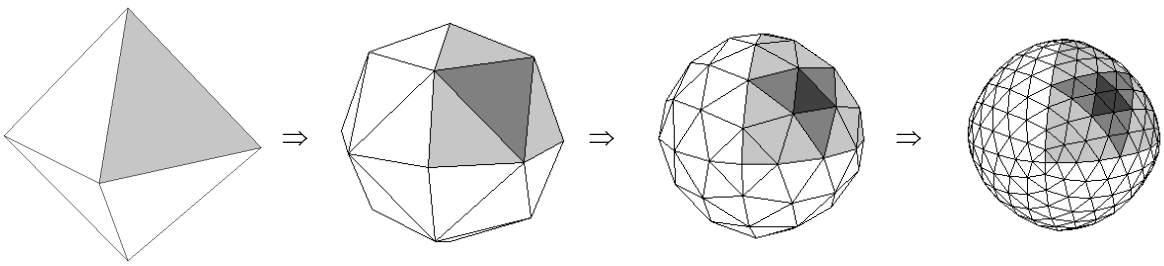


Figure 3 — Nested control meshes

Figure 3 shows an example of how subdivision would be applied to a triangular mesh. At each step, one triangle is shaded to highlight the correspondence between one level and the next: each triangle is broken down into four new triangles, whose vertex positions are perturbed to have a smoother and smoother surface.

3.3.2.1.1 Piecewise Smooth Surfaces

Subdivision schemes for piecewise smooth surfaces include common modeling primitives such as quadrilateral free-form patches with creases and corners.

A somewhat informal description of piecewise smooth surfaces is given here; for simplicity, only surfaces without self intersections are considered. For a closed C^1-continuous surface in $\Re^3$, each point has a neighborhood that can be smoothly deformed into an open planar disk *D*. A surface with a *smooth boundary* can be described in a similar way, but neighborhoods of boundary points can be smoothly deformed into a half-disk *H*, with closed boundary (Figure 4). In order to allow *piecewise* smooth boundaries, two additional types of local charts are introduced: concave and convex corner charts, Q_3 and Q_1.

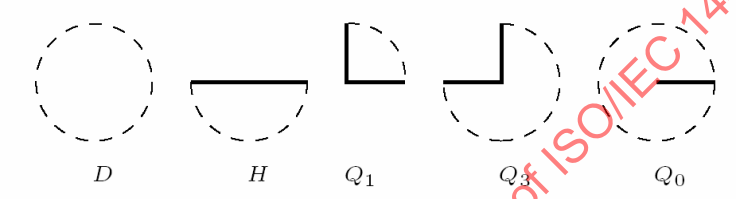


Figure 4 — The charts for a surface with piecewise smooth boundary

Piecewise-smooth surfaces are constructed out of surfaces with piecewise smooth boundaries joined together. If two surface patches have a common boundary, but different normal directions along the boundary, the resulting surface has a sharp crease.

Two adjacent smooth segments of a boundary are allowed to be joined, producing a crease ending in a *dart* (cf. [37]). For dart vertices an additional chart Q_0 is required; the surface near a dart can be deformed into this chart smoothly everywhere except at an open edge starting at the center of the disk.

3.3.2.1.2 Tagged meshes

Before describing the set of subdivision rules, the description of the tagged meshes is described, which the algorithms accept as input. These meshes represent piecewise-smooth surfaces with features described above.

The complete list of tags is as follows. Edges can be tagged as *crease edges*. A vertex with incident crease edges receives one of the following tags:

- Crease vertex: joins exactly two incident crease edges smoothly.
- Corner vertex: connects two or more creases in a corner (convex or concave).
- Dart vertex: causes the crease to blend smoothly into the surface; this is the only allowed type of crease termination at an interior non-corner vertex.

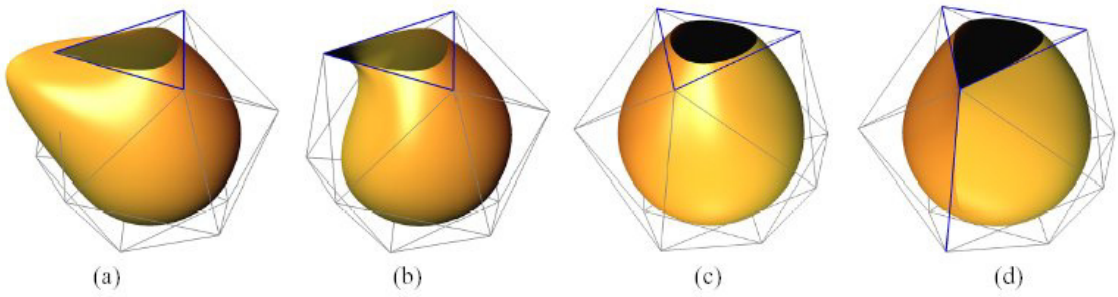


Figure 5 — Features: (a) concave corner, (b) convex corner, (c) smooth boundary/crease, (d) corner with two convex sectors.

Boundary edges are automatically assigned crease tags; boundary vertices that do not have a corner tag are assumed to be smooth crease vertices.

Crease edges divide the mesh into separate patches, several of which can meet in a corner vertex. At a corner vertex, the creases meeting at that vertex separate the ring of triangles around the vertex into sectors. Each sector of the mesh is labeled as *convex sector* or *concave sector* indicating how the surface should approach the corner.

In all formulas *k* denotes the number of faces incident at a vertex in a given sector. Note that this is different from the standard definition of the vertex degree: faces, rather than edges are counted. Both quantities coincide for interior vertices with no incident crease edges. This number is referred as *crease degree*.

The only restriction on sector tags is that concave sectors consist of at least two faces. An example of a tagged mesh is given in Figure 6.

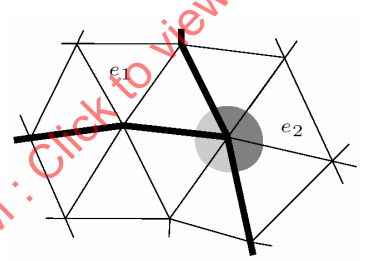


Figure 6 — Crease edges meeting in a corner with two convex (light grey) and one concave (dark grey) sectors. The subdivision scheme modifies the rules for edges incident to crease vertices (e.g. e_1) and corners (e.g. e_2).

3.3.2.2 SubdivisionSurface

3.3.2.2.1 Node interface

SubdivisionSurface { #%NDT=SFGeometryNode, SFBaseMeshNode

eventIn	MFInt32	set_colorIndex		
eventIn	MFInt32	set_coordIndex		
eventIn	MFInt32	set_texCoordIndex		
eventIn	MFInt32	set_creaseEdgeIndex		
eventIn	MFInt32	set_cornerVertexIndex		
eventIn	MFInt32	set_creaseVertexIndex		
eventIn	MFInt32	set_dartVertexIndex		
exposedField	SFColorNode	color	NULL	
exposedField	SFCoordinateNode	coord	NULL	
exposedField	SFTextureCoordinateNode	texCoord	NULL	
exposedField	SFInt32	subdivisionType	0	# 0 – midpoint, 1 – Loop, 2 – butterfly, 3 – Catmull-Clark
exposedField	SFInt32	subdivisionSubType	0	# 0 – Loop, 1 – Warren-Loop
field	MFInt32	invisibleEdgeIndex	[]	# [0, ∞)
# extended Loop if non-empty				
exposedField	SFInt32	subdivisionLevel	0	# [-1, ∞)
# IndexedFaceSet fields				
field	SFBool	ccw	TRUE	
field	MFInt32	colorIndex	[]	# [-1, ∞)
field	SFBool	colorPerVertex	TRUE	
field	SFBool	convex	TRUE	
field	MFInt32	coordIndex	[]	# [-1, ∞)
field	SFBool	solid	TRUE	
field	MFInt32	texCoordIndex	[]	# [-1, ∞)
# tags				
field	MFInt32	creaseEdgeIndex	[]	# [-1, ∞)
field	MFInt32	dartVertexIndex	[]	# [-1, ∞)
field	MFInt32	creaseVertexIndex	[]	# [-1, ∞)
field	MFInt32	cornerVertexIndex	[]	# [-1, ∞)
# sector information				
exposedField	MFSubdivSurfaceSectorNode	sectors	[]	

}

3.3.2.2.2 Functionality and semantics

The **SubdivisionSurface** node is similar to the existing **IndexedFaceSet** node with all fields relating to face normals and the **creaseAngle** field removed since normals are generated by the subdivision algorithm at run-time and their behavior at facet boundaries is controlled by edge and vertex crease tagging. The control hull is specified as a collection of polygons and must be a manifold mesh. No “dangling” edges or vertices are permitted, and each polygon may share each of its edges with at most one other polygon. More details about the subdivision process and rules are described in 3.3.2.2.3.

The **SubdivisionSurface** node represents a 3D shape formed through surface subdivision of a control hull constructed from vertices listed in the **coord** field. The **coord** field contains a **Coordinate** node that defines the 3D vertices referenced by the **coordIndex** field. **SubdivisionSurface** uses the indices in its **coordIndex** field to specify the polygonal faces of the control hull by indexing into the coordinates in the **Coordinate** node. An index of “-1” is used as a separator for the control hull faces. The last face may be (but does not have to be) followed by a “-1” index. If the greatest index in the **coordIndex** field is N, then the **Coordinate** node shall contain N+1 coordinates (indexed as 0 to N). Each face of the **SubdivisionSurface** control hull shall have:

- exactly three non-coincident vertices for triangular schemes (midpoint, Loop and butterfly);
- at least three non-coincident vertices for quadrilateral schemes (Catmull-Clark and extended Loop);
- vertices that define a planar, non-self-intersecting polygon.

Otherwise, the results are undefined.

The **SubdivisionSurface** node is specified in the local coordinate system and is affected by the transformation of its ancestors.

The **subdivisionType** field specifies the major subdivision scheme:

1) Midpoint (non-smoothing);
2) Loop (subclause 3.3.2.2.3.1);
3) Dyn's butterfly (cf. [31], [91]);
4) Catmull-Clark (subclause 3.3.2.2.3.3).

The **subdivisionSubType** field specifies a sub category of subdivision rules in the major subdivision rule. For this specification, for **subdivisionType** 0 (Loop), **subdivisionSubType** 0 indicates Loop and 1 Warren's Loop.

A **subdivisionType** value of zero and a non-empty **invisibleEdgeIndex** array imply that the extended Loop scheme (subclause 3.3.2.2.3.2) is used.

The **invisibleEdgeIndex** field is used to select co-ordinate pairs from **coord** to define which edges should be tagged as invisible. Consecutive pairs of co-ordinates define an edge; this pair of vertices must belong to one facet as defined by the **coordIndex** field and may be "interior" to the facet (e.g. in the case of a quadrilateral, the invisible edge would be one of its diagonals). All non-triangular facets should have enough tags to fully describe their triangulation. Note that a non-empty **invisibleEdgeIndex** array is only allowed when **subdivisionType** is 0; the results are undefined in other cases.

The **subdivisionLevel** field specifies the number of times the surface should be subdivided. If this field is "-1" then the terminal should apply enough iterations of subdivision until it reaches a "visually smooth" surface.

The fields **color**, **coord**, **texCoord**, **ccw**, **colorIndex**, **colorPerVertex**, **convex**, **coordIndex**, **solid**, and **texCoordIndex** have the same semantic as for the **IndexedFaceSet** node. These fields define the control hull of the subdivision surface. Textures coordinates and color information, if present, propagate from the control hull to subsequent subdivision levels through linear interpolation.

The **creaseEdgeIndex** field containsan indexed line set of creases, indicating where creases appear on the otherwise smooth subdivision surface (subclause 3.3.2.1.2). As for the **IndexedLineSet** node, the line is defined by successive vertex indices defined in the **coord** field. An index of "-1" indicates that the current crease has ended and the next one begins. The last crease may be (but does not have to be) followed by a "-1". Crease edges must already exist in the control hull topology (i.e. the edge must coincide with a single edge of a polygon as defined by the **coordIndex** field). Note that if an edge is tagged as both invisible and a crease, the results are undefined.

From the **creaseEdgeIndex** array, the default behavior (and appropriate subdivision rules) for vertices on crease edges are defined (subclause 3.3.2.1.2) as:

- Vertices with exactly one crease edge in their neighborhood are dart vertices,
- Vertices with exactly two crease edges in their neighborhood are crease vertices,
- Vertices with three or more crease edges in their neighborhood are corner vertices.

However, this array may not provide a description general enough for all surfaces and three arrays are provided: **cornerVertexIndex**, **creaseVertexIndex**, and **dartVertexIndex**. The information provided by these arrays completes (or overrides) the information in **creaseEdgeIndex**.

The **cornerVertexIndex** field indicates the vertex index of corner vertices.

The **creaseVertexIndex** field indicates the vertex index of crease vertices.

The **dartVertexIndex** field indicates the vertex index of dart vertices.

Finally, the **sectors** field may optionally contain an array of **SubdivSurfaceSector** nodes that describe additional sector information; see subclause 3.3.2.2.3 for more details. Note that if both the **sector** and the **invisibleEdgeIndex** fields are non-empty, the results are undefined.

3.3.2.2.3 Subdivision Stencils

In order to reduce the angles between adjacent triangles, the position of each new vertex introduced by the subdivision process is calculated as a function of some old vertices in the vicinity of the considered edge. The position of old vertices themselves may be altered as well for that same purpose.

The *stencil* of the subdivision scheme refers to the set of old vertices used to compute a new vertex. The term *mask,* or the set of (averaging) rules, is used to refer to the set of values of coefficients. It determines the convergence of the process and the smoothness degree of the limit surface.

Different masks are necessary for different types of vertices discussed in more detail in the next subclause.

3.3.2.2.3.1 Loop subdivision

Figure 7 shows the stencils used for repositioning old vertices and inserting new ones by splitting edges in the case of Loop's scheme [48].

Vertex rules are used to update positions of old vertices, edge rules are used to compute positions of newly inserted vertices.

For untagged interior and dart vertices rules shown in Figure 7a are used; for boundary/crease vertices rules in Figure 7b are used. Positions of corner vertices are never changed.

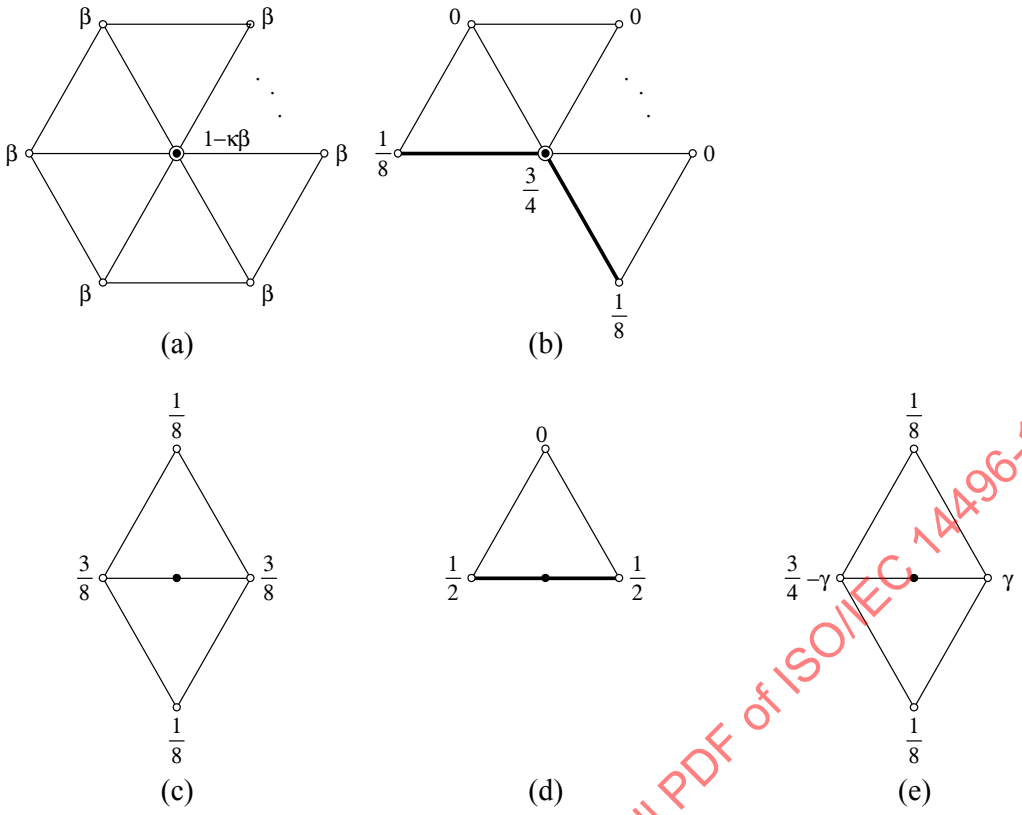


Figure 7 — Stencils of Loop's subdivision scheme. (a) repositioning interior old vertices, (b) repositioning boundary or crease old vertices, (c) splitting interior edges not adjacent to boundary extraordinary vertices, (d) splitting boundary and crease edges, (e) splitting interior edges adjacent to an extraordinary (valence not 4) vertex on the boundary/crease or corner vertex. The values of beta and gamma are defined in the text.

The coefficient β can be chosen in different ways.

Loop's original proposal was $\beta = \frac{1}{k}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right)$. In particular, this is β = 3/16 for k = 3, and β = 1/16 for k = 6. Warren proposed later to keep β = 3/16 for k = 3, but to have simply $\beta = 3/8k$ for k > 3, which also amounts to β = 1/16 for k = 6 [84].

Edge rules are more complex. The rule for an edge point inserted on an edge *e* is chosen depending on the tag of the edge and the tags of adjacent vertices and sectors. There are several cases:

- In the absence of tags, the standard edge rules are applied (Figure 7c).
- If a crease edge e is split, and no endpoint is a dart vertex, the rule shown in Figure 7d is used.
- If one of the endpoints is a dart vertex, the standard rule (Figure 7c) is applied.
- If one endpoint v is an extraordinry vertex (valence not 4) and tagged as corner or smooth crease and the edge e is not a crease edge, we use a modified rule shown in Figure 7e (with weight $3/4 - \gamma$ associated with extraordinary vertex), with parameter γ chosen as $\gamma(\theta_k) = 1/2 - 1/4\cos\theta_k$. For a crease vertex, $\theta_k = \pi/k$, where k is the crease degree of the vertex in the sector where the edge e is.

- If both endpoints are extraordinary vertices (valence not 4) and tagged as corner or smooth crease and the edge e is not a crease edge, then the average of the masks given by the rules in 4 above is taken.

At a corner vertex *v*, the edge *e* is differentiated whether it is in a convex or concave sector. For a convex corner, $\theta_k = \alpha / k$, where α is the angle between the two crease edges spanning the sector, for concave corners, $\theta_k = (2\pi - \alpha)/k$.

For concave corners, one may use $\theta_k > \pi/k$ and for convex corners $\theta_k < \pi/k$. The simplest choice of θ_k is $3\pi/(2k)$ for concave corners and $\pi/(2k)$ for convex corners which is the default; the best choice is to use α/k for convex corners (respectively $(2\pi-\alpha)/k$ for concave corners) where α is the angle between edges of the control mesh bounding the sector; θ_k for concave and convex corners can be also a user-defined parameter changing in the ranges specified above.

Application of the stencils is followed by application of flatness modification described in the next subclause. This modification is optional for all cases excluding concave corner; in which case it is necessary to ensure tangent plane continuity of the surface.

- **Flatness modification**

With *k* defined as in previous subclause, let the vector of control points be $p^m = \{p_c^m, p_0^m, \ldots, p_{k-1}^m\}$ (Figure 8). The superscript indicates the subdivision level the subscript indicates the point number in the sector. After subdivision step in a neighborhood of a corner vertex is modified as follows:

$$p^{new} = (1-s)p + s(\mathbf{a}_0 x^0 + \mathbf{a}_1 x^1 + \mathbf{a}_2 x^2)$$

where *s* is the flatness parameter. $\mathbf{a}_i = (l^i, p), l^0, l^1, l^2$ are limit position and tangent masks defined in the next subclause, and x^0, x^1, x^2 are the right eigenvectors also defined in the next subclause. The valid range of the flatness parameter is $0 \le s \le 1$ for all vertex types except concave corner, in which case it is $1 - \frac{1}{2\lambda} < s \le 1$, with lambda defined as $\lambda = 1/2 - 1/4(\cos\theta_k - \cos(\pi/k))$. Geometrically, the modified rule blends between control point positions before flatness modification and certain points in the tangent plane, which are typically close to the projection of the original control point. The limit position $\mathbf{a}_0$ of the center vertex remains unchanged.

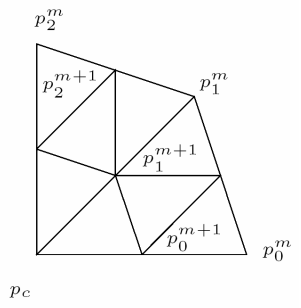


Figure 8 — Neighborhoods of a vertex on different subdivision levels. The subdivision matrix relates the vector of control points p^m to the control points on the next level p^{m+1}. For a neighborhood of k triangles $p^m = \{p_c^m, p_0^m, \ldots, p_{k-1}^m\}$.

The flatness modification is always applied at concave corner vertices; the default value for the flatness parameter is 1. Better visual results are obtained for $s = 1 - 1/(4\lambda)$. In other cases, s can be taken to be 0 by default and the modification step need not be applied.

Optionally, a user-specified flatness value can be associated with any vertex and used to control how fast the surface approaches the tangent plane.

A compliant implementation is required to support *s = 0* and *s = 1*, but an implementation may support arbitrary *s* values.

- **Limit position and tangent masks**

For each type of vertex (interior, smooth crease, corner, dart) there are masks for computing the limit position of this vertex and two tangent vectors at this vertex. The limit position masks are denoted l^0 and limit tangent position masks are denoted l^1 and l^2 respectively. A subscript *c* denotes the coefficient corresponding to the center vertex. Once two tangent vectors are obtained by applying the masks, the normal can be computed as the cross-product of the two. In addition we define right eigenvectors in each case which are necessary for flatness modification. Recall that dominant right eigenvector x^0 is the vector consisting of ones

Untagged interior or dart vertex of degree k. In all cases i is in the range $\theta = k - 1$ and $\theta_k = 2\pi / k$.

$$l_c^0 = \frac{1}{1+(8k/3)\beta}, \quad l_i^0 = \frac{(8/3)\beta}{1+(8k/3)\beta}$$

$$x_c^1 = x_c^2 = l_c^1 = l_c^2 = 0$$

$$x_i^1 = \sin i\theta_k, \quad x_i^2 = \cos i\theta_k,$$

$$l_i^1 = \frac{2}{k}\sin i\theta_k, \quad l_i^2 = \frac{2}{k}\cos i\theta_k,$$

Smooth crease vertex of crease degree k. Let $\theta_k = \pi / k$

$$l_c^0 = 2/3, \quad l_1^0 = l_k^0 = 1/6, \quad l_i^0 = 0, \quad i = 1 \ldots k-1$$

For k=1,

$$x_c^1 = -1/3, x_1^1 = x_2^1 = 2/3, x_c^2 = 0, x_1^2 = 1, x_2^2 = -1$$

$$l_c^1 = -1, l_1^1 = l_2^1 = 1/2, l_c^2 = 0, l_1^2 = 1/2, l_2^2 = -1$$

Otherwise $x_c^1 = x_c^2 = l_c^1 = 0$,

$$x_i^1 = \sin i\theta_k, x_i^2 = \cos i\theta_k, i = 0 \ldots k$$

$$l_0^1 = 1/2, l_k^1 = -1/2, l_i^1 = 0, i = 1 \ldots k$$

$$l_c^2 = -\frac{2}{k}\left(\left(\frac{2}{3} - a\right)\sigma_1 - b\sigma_3\right)$$

$$l_0^2 = l_k^2 = -\frac{2}{k}\left(\left(\frac{a}{2} + \frac{1}{6}\right)\sigma_1 + \frac{1}{2}b\sigma_3\right)$$

$$l_i^2 = \frac{2}{k}\sin i\theta_k, \quad i = 1 \ldots k-1$$

where

$$a = \frac{\frac{1}{4}(1+\cos\theta_k)}{3(\frac{1}{2}-\frac{1}{4}\cos\theta_k)} \qquad b = \frac{\frac{2}{3}-a}{\cos\frac{k\zeta}{2}}$$

$$\sigma_1 = \frac{\sin\theta_k}{1-\cos\theta_k} \qquad \sigma_3 = \frac{\cos\frac{k\zeta}{2}\sin\theta_k}{\cos\zeta-\cos\theta_k}$$

$$\zeta = \arccos(\cos\theta_k - 1)$$

Convex/concave corner vertex of crease degree k with parameter θ_k. Let $\theta = k\theta_k$.

$$l_c^0 = 1, l_i^0 = 0, i = 0\ldots k$$

$$x_c^1 = x_c^2 = 0, x_i^1 = \frac{\sin i\theta_k}{\sin\theta}, x_i^2 = \frac{\sin(k-i)\theta_k}{\sin\theta}, i = 0\ldots k$$

$$l_c^1 = -1, l_k^1 = 1, l_i^1 = 0, i = 0\ldots k-1$$

$$l_c^2 = -1, l_0^2 = 1, l_i^2 = 0, i = 1\ldots k$$

3.3.2.2.3.2 Extended Loop subdivision

Extended Loop is an enhancement of the well-known triangle-based Loop subdivision algorithm that allows the designer to work with quadrilateral-based models which are better suited to capturing the symmetries of natural and man-made objects, while ensuring that the visualisation process can be optimised for a triangle-only rendering pipeline. Furthermore, unlike quadrilateral-based subdivision schemes such as Catmull-Clark, extended Loop offers the model designer exact control over the final model triangulation, which is essential for efficient rendering of low polygon models.

- **Stencils**

Extending Loop to handle arbitrary meshes requires the addition of edge visibility tags for edges that are introduced during triangulation and are not part of the original control hull. The introduction of edge visibility information will clearly affect the vertex and edge subdivision rules.

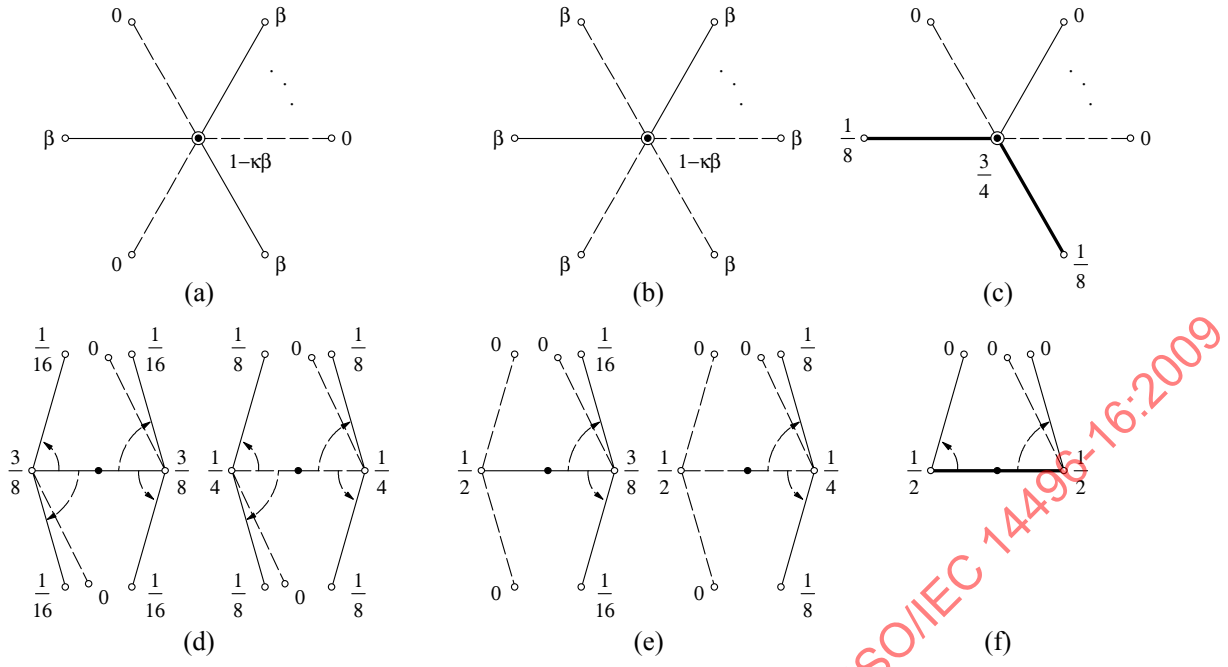


Figure 9 — Stencils of extended Loop subdivision for repositioning interior(smooth and dart) (a), (b) and boundary/crease old vertices and splitting interior (d), (e) and boundary/crease edges.

Figure 9 shows the extended Loop subdivision stencils used for repositioning old vertices (as with standard Loop, corner vertices are never moved) and splitting edges both on the surface interior and boundaries/creases. Solid lines indicate original control hull (visible) edges while dashed lines indicate invisible edges. It is clear from the stencils (a), (b) and (c) that repositioning old vertices with extended Loop is identical to standard Loop if invisible edges are ignored, with one exception (b). If a vertex has two or less visible edges in its neighborhood, then the invisible edges are included in the calculation. If all edges are visible then the stencils are identical to standard Loop subdivision.

As with standard Loop subdivision, $\beta = \frac{1}{k}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right)$, where k is the number of visible edges (Figure 9a) or the valence (visible + invisible edges) if the number of visible edges is ≤ 2 (Figure 9b).

The stencil for splitting boundary/crease edges (Figure 9f) is also identical to standard Loop, while splitting interior edges depends on whether the edge in question is visible (Figure 9d – left) or not (Figure 9d – right). As with the old vertex repositioning stencils, the edge split stencils try to employ vertices at the "other end" of visible edges only, ignoring invisible edges completely. Identifying visible edges at each side of the edge endpoint involves a clockwise and/or counter-clockwise search around the endpoint neighborhood and may result in the same vertex for both sides or no visible edges in which case the endpoint is assigned the full weight (Figure 9e). Again if all edges are visible, the stencils are identical to standard Loop subdivision.

- **Visibility propagation**

The visibility of the new edges needs to be determined, since it will be required on the next round of subdivision. New edges radiate from a new point (see Figure 10), and their visibility is determined according to the neighborhood of the point.

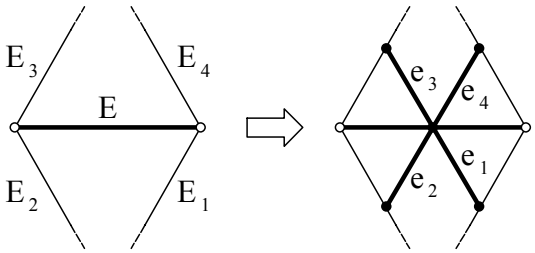


Figure 10 — New edges created when splitting an edge in two.

Two edges will be the two halves of the original edge E - these simply keep the visibility of the original edge. Of the other four new edges, two edges will be to the right hand side of the original edge, and two more to its left. If we label these edges clockwise as e1, e2, e3 and e4, these connect to other new points created by dividing the original edges labeled E1, E2, E3 and E4. The visibility of the new edges is initialized to invisible and is changed to visible if the lookup of the combination of E, E1, E2, E3 and E4 in Table 1 indicates that it should be. Clearly, the visibility of each of e1, e2, e3 and e4 is not uniquely determined by splitting E alone; e1, for example, also depends on the visibility lookup for splitting E1.

Table 1 — New edge visibility lookup table

Original edges visible					New edges visible				Original edges visible					New edges visible			
E	E_1	E_2	E_3	E_4	e_1	e_2	e_3	e_4	E	E_1	E_2	E_3	E_4	e_1	e_2	e_3	e_4
									✓								
				✓	✓		✓		✓				✓				
			✓			✓			✓			✓					
			✓	✓	✓	✓			✓			✓	✓			✓	✓
		✓			✓		✓		✓		✓						
		✓		✓	✓		✓		✓		✓		✓				
		✓	✓		✓			✓	✓		✓	✓					
		✓	✓	✓	✓				✓		✓	✓	✓			✓	✓
	✓					✓		✓	✓	✓							
	✓			✓		✓	✓		✓	✓			✓				
	✓		✓			✓		✓	✓	✓		✓					
	✓		✓	✓		✓			✓	✓		✓	✓			✓	✓
	✓	✓					✓	✓	✓	✓	✓			✓	✓		
	✓	✓		✓			✓		✓	✓	✓		✓	✓	✓		
	✓	✓	✓					✓	✓	✓	✓	✓		✓	✓		
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Note that Table 1 is unchanged if we swap left and right, so it is unimportant which side of the original edge we call “right” to start with.

- **Vertex normal calculation**

In order to perform operations such as lighting on a surface, it is necessary to determine the normal to that surface at each vertex. This is the direction at right angles to the limit surface at that point. In order to calculate this normal, it is necessary first to obtain two tangent vectors, t_1 and t_2, which are directions that lie along the surface rather than at right angles to it. Given these, a vector cross product will result in the desired normal:

$$n = t_1 \times t_2.$$

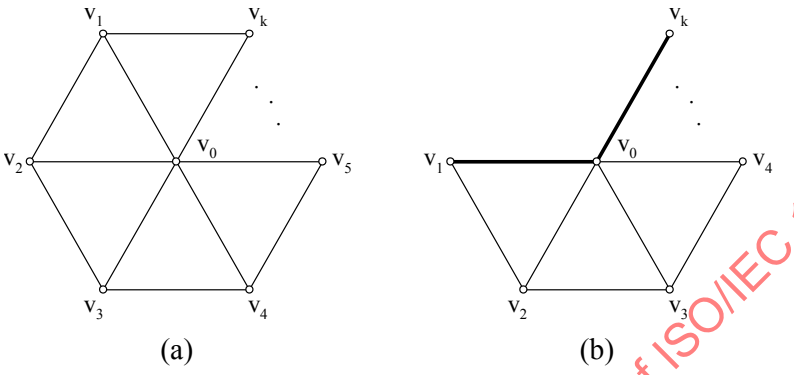


Figure 11 — (a) interior, and (b) boundary/crease vertex neighborhood used for normal calculation.

For interior smooth and dart vertices of valence k (Figure 11a), the tangents are defined as the weighted averages of the positions of the neighborhood vertices (v_i where $i \in [1, k]$):

$$t_1 = c_1 v_1 + c_2 v_2 + c_3 v_3 + \ldots + c_k v_k$$
$$t_2 = c_2 v_1 + c_3 v_2 + c_4 v_3 + \ldots + c_k v_{k-1} + c_1 v_k,$$

where:

$$c_i = \cos\frac{2\pi i}{k}.$$

For boundary, crease and corner vertices (Figure 11b), each sector between pairs of boundary/crease edges will have a different pair of tangent vectors that are defined as:

$$t_1 = v_1 - v_k$$
$$t_2 = \begin{cases} -2v_0 + v_1 + v_2, & k = 2 \\ v_2 - v_0, & k = 3 \\ -2v_0 - v_1 + 2v_2 + 2v_3 - v_4, & k = 4 \\ w_1 v_1 + w_2 v_2 + w_3 v_3 + \ldots + w_k v_k, & k > 4 \end{cases},$$

where, for $\theta = \frac{\pi}{k-1}$, w_i is defined as:

$$w_i = \begin{cases} \sin\theta, & i = 1, i = k \\ (2\cos\theta - 2)\sin((i-1)\theta), & 2 \le i \le k-1 \end{cases}.$$

3.3.2.2.3.3 Catmull-Clark subdivision

The Catmull-Clark scheme was described in [19]. This scheme can be applied to general polygonal meshes, but after the first subdivision step all polygons in a mesh become quads. The masks for a mesh consisting only of quads are shown in Figure 12 and Figure 13. The scheme produces surfaces that are C^2 everywhere except at extraordinary vertices, where they are C^1.

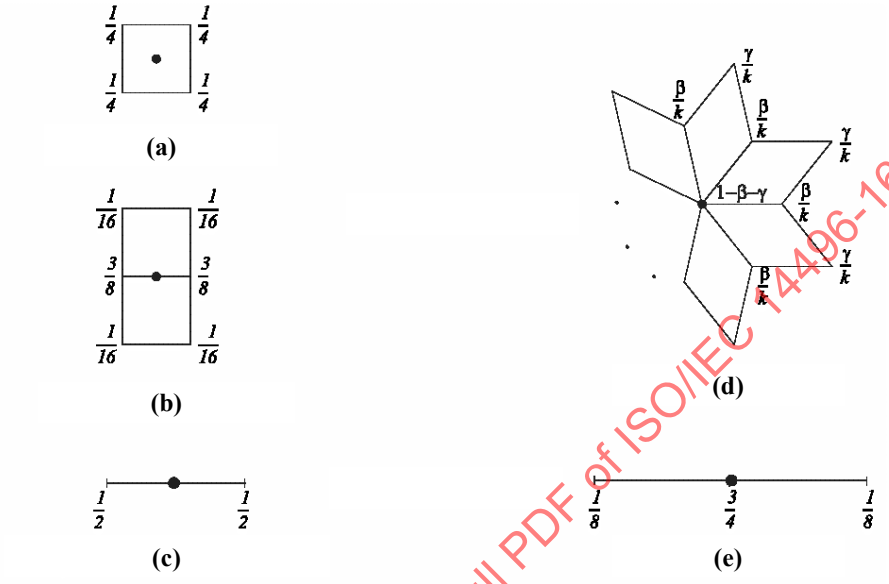


Figure 12 — Catmull-Clark rules with k defined as in subclause 3.3.2.1.2 (a) face rule; (b) regular edge rule (c) boundary/crease edge rule (d) interior vertex rule $\beta = 3/2k$ and $\delta = 1/4k$ (e) smooth boundary/crease vertex rule

There are three types of rules: vertex rules used to update old vertex positions, edge rules used to insert new vertices for each edge, and face rules used to insert new vertices for each face.

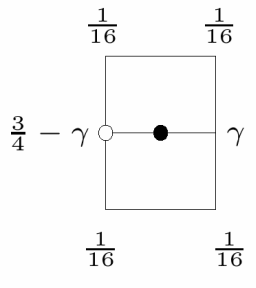


Figure 13 — Modified rule for odd vertices adjacent to a boundary extraordinary vertex (Catmull-Clark scheme), the coefficient gamma is defined in the text.

The rules used only on the first step are:

- a face control point for an n-gon is computed as the average of the corners of the polygon;
- an edge control point for an untagged edge as the average of the endpoints of the edge and newly computed face control points of adjacent faces; for a crease/boundary edge it is the average of the endpoints;

- The vertex rule for untagged interior and dart vertices is:

$$p_c^{j+1} = \frac{k-2}{k} p_c^j + \frac{1}{k^2}\left(\sum_{i=0}^{k-1} p_i^j + q_i^{j+1}\right)$$

where p_i^j are k control points at level j sharing edges with p_c^j, and q_c^{j+1} are face control points for faces incident at the vertex. Note that face control points on level j+1 are used.

- For boundary/crease vertices, it is the spline rule identical to the one used for the Loop scheme. The corner vertex positions are never changed.

After the first subdivision step the mesh consists only of quadrilaterals and the subdivision rules are described in (Figure 12 and Figure 13). The vertex and face rules are essentially the same; the masks are shown in Figure 12. As in the case of the Loop scheme edge rules are more complex:

- In the absence of tags, the standard edge rules are applied (Figure 14b).
- If a crease edge e is split and no endpoint is tagged as a dart vertex , rules of Figure 14c are used.
- If one of the endpoints is a dart vertex, the standard edge rule is used (Figure 14b).
- If one endpoint v is tagged as a corner or smooth crease and the edge e is not a crease, a modified rule shown in Figure 6 is used, with parameter γ chosen as $\gamma(\theta_k) = 3/8 - 1/4\cos\theta_k$. Exactly like for Loop scheme, for a smooth crease vertex, $\theta_k = \pi/k$, where k is the crease degree of v in the sector where e is. For a corner use $\theta_k = \alpha/k$, where $\alpha > \pi$ for concave corners and $\alpha < \pi$ for convex corners.
- If both endpoints are tagged as corner or smooth crease, then the average of the masks given by the rules above is taken.

Application of the stencils is followed by application of flatness modification described in the next subclause. This modification is optional for all cases excluding concave corner; in this case it is necessary to ensure tangent plane continuity of the surface.

- **Flatness modification**

The flatness modification is similar to the one used for Loop scheme, but the vector of control points has to include one more group of points $[q_0^m, \ldots, q_{k-1}^m]$ as shown in Figure 14. The superscript indicates the subdivision level the subscript indicates the point number in the sector.

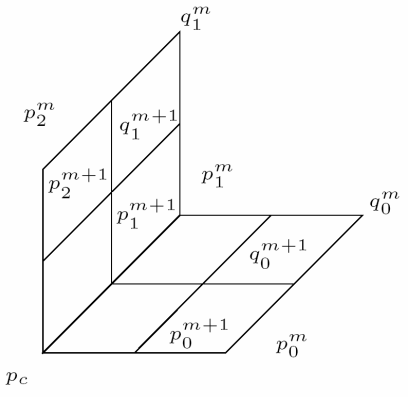


Figure 14 — Neighborhoods of a vertex on different subdivision levels. The subdivision matrix relates the vector of control points p^m to the control points on the next level p^{m+1}. For a neighborhood of k quadrilaterals $p^m = \{p_c^m, p_0^m, \ldots, p_{k-1}^m, q_0^m, \ldots, q_{k-1}^m\}$.

The rule itself coincides with the rule for the Loop scheme:

$$p^{new} = (1-s)p + s(\mathbf{a}_0 x^0 + \mathbf{a}_1 x^1 + \mathbf{a}_2 x^2)$$

where *s* is the flatness parameter. $\mathbf{a}_i = (l^i, p), l^0, l^1, l^2$ are limit position and tangent masks defined in the next subclause, and x^0, x^1, x^2 are the right eigenvectors also defined in the next subclause.

The valid range of the flatness parameter is $0 \le s \le 1$ for all vertex types except concave corner, in which case it is $1 - \frac{1}{2\lambda} < s \le 1$, with lambda defined as

$$\lambda = \frac{1}{2}\gamma + \frac{1}{8}c^2 + \frac{1}{16}\sqrt{64\gamma^2 + 32c^2\gamma - 48\gamma + 4c^4 + 4c^2 + 9}$$

and

$$c = \cos\frac{\pi}{2k}$$

The default flatness parameter can be set to zero for all cases except concave corners, in which case it is set to 1. Better visual results are obtained for the values close to $s = 1 - 1/(4\lambda)$.

Optionally, a user-specified flatness value can be associated with any vertex and used to control how fast the surface approaches the tangent plane.

- **Limit position and tangent masks**

Interior or dart vertex of degree k. In all cases i is in the range $0 \ldots k-1$, and $\theta_k = 2\pi/k$.

$$l_c^0 = \frac{k}{k+5}, \quad l_{pi}^0 = \frac{4}{k(k+5)}, \quad l_{qi}^0 = \frac{1}{k(k+5)}$$

$$x_c^1 = x_c^2 = l_c^1 = l_c^2 = 0$$

$$x_{pi}^1 = \frac{1}{\sigma}\sin i\theta_k \quad x_{pi}^2 = \frac{1}{\sigma}\cos i\theta_k$$

$$x_{qi}^1 = \frac{\sin i\theta_k + \sin(i+1)\theta_k}{\sigma(4\lambda-1)}, \quad x_{qi}^2 = \frac{\cos i\theta_k + \cos(i+1)\theta_k}{\sigma(4\lambda-1)},$$

$$l_{pi}^1 = 4\sin i\theta_k, \quad l_{pi}^2 = 4\cos i\theta_k,$$

$$l_{qi}^1 = \frac{\sin i\theta_k + \sin(i+1)\theta_k}{4\lambda-1}, \quad l_{qi}^2 = \frac{\cos i\theta_k + \cos(i+1)\theta_k}{4\lambda-1}$$

where $\lambda = 5/16 + 1/16(\cos\theta_k + \cos\theta_k / 2\sqrt{9+\cos 2\theta_k})$ and $\sigma = k(2 + \frac{1+\cos\theta_k}{(4\lambda-1)^2})$

Smooth crease vertex of crease degree k. Let $\theta_k = \pi / k$

$$l_c^0 = 2/3, \quad l_{p1}^0 = l_{pk}^0 = 1/6,$$
$$l_{q0}^0 = l_{qi}^0 = l_{pi}^0 = 0, \quad i = 1 \ldots k-1$$

For $k = 1$,

$$x_c^1 = 1/18 \quad x_{p0}^1 = -2/18 \quad x_{p1}^1 = -2/18 \quad x_{q0}^1 = -5/18$$
$$x_c^2 = 0 \quad x_{p0}^2 = -1/2 \quad x_{p1}^2 = 1/2 \quad x_{q0}^2 = 0$$
$$l_c^1 = 6 \quad l_{p0}^1 = -3 \quad l_{p1}^1 = -3 \quad l_{q0}^1 = 0$$
$$l_c^2 = 0 \quad l_{p0}^2 = -1 \quad l_{p1}^2 = 1 \quad l_{q0}^2 = 0$$

Otherwise $l_c^2 = x_c^1 = x_c^2 = 0,$

$$x_{pi}^1 = \sin i\theta_k, x_{pi}^2 = \cos i\theta_k, i = 0 \ldots k$$
$$x_{qi}^1 = \sin i\theta_k + \sin(i+1)\theta_k, i = 0 \ldots k-1$$
$$x_{qi}^2 = \cos i\theta_k + \cos(i+1)\theta_k, i = 0 \ldots k-1$$
$$l_{p0}^2 = 1/2, l_{pk}^2 = -1/2, l_{q0}^2 = l_{qi}^2 = l_{pi}^2 = 0, i = 1 \ldots k-1$$
$$l_c^1 = 4R(\cos\theta_k - 1), l_{p0}^1 = l_{pk}^1 = -R(1+2\cos\theta_k)$$
$$l_{pi}^1 = \frac{4\sin i\theta_k}{(3+\cos\theta_k)k}, i = 1 \ldots k-1$$
$$l_{qi}^1 = \frac{4(\sin i\theta_k + \sin(i+1)\theta_k)}{(3+\cos\theta_k)k}, i = 0 \ldots k-1$$

where $R = \frac{\cos\theta_k + 1}{k\sin\theta_k(3+\cos\theta_k)}$

Convex/concave corner vertex of crease degree k with parameter θ_k. Let $\theta = k\theta_k$.

Left eigenvectors are the same as for Loop with zeroes everywhere except lc, lp0 and lpk.

$$x^1_{pi} = \frac{\sin i\theta_k}{\sin\theta}, x^2_{pi} = \frac{\sin(k-i)\theta_k}{\sin\theta}, i = 0\ldots k$$

$$x^1_{qi} = \frac{\sin i\theta_k + \sin(i+1)\theta_k}{\sin\theta}, i = 0\ldots k-1$$

$$x^2_{qi} = \frac{\sin(k-i)\theta_k + \sin(k-i-1)\theta_k}{\sin\theta}, i = 0\ldots k-1$$

3.3.2.2.3.4 Normal control

In addition to the flatness modification which insures that the surface is tangent plane continuous at concave corner vertices and provides additional shape control at other vertex types, a similar modification mechanism can be used to control the tangent plane of the surface as shown in Figure 15.

Specifically, if the user prescribes tangent vectors or the normal for a sector at a corner or smooth boundary vertex or a normal for an interior vertex then the following modification can be applied after each subdivision step for both Loop and Catmull-Clark surfaces, $p^{new} = p + t((\mathbf{a}'_1 - \mathbf{a}_1)x^1 + (\mathbf{a}'_2 - \mathbf{a}_2)x^2)$ where the notation used is the same as in the subclauses describing flatness modification above, $\mathbf{a}'_1$ and $\mathbf{a}'_2$ denote the prescribed tangents and t is a user-specified parameter $0 \le t \le 1$, by default set to 1. If only the normal is prescribed, the tangents are computed as $a'_i = a_i - (a_i, n)n$ for $i = 1,2$.

The parameter t determines how fast the surface approaches the prescribed tangent plane.

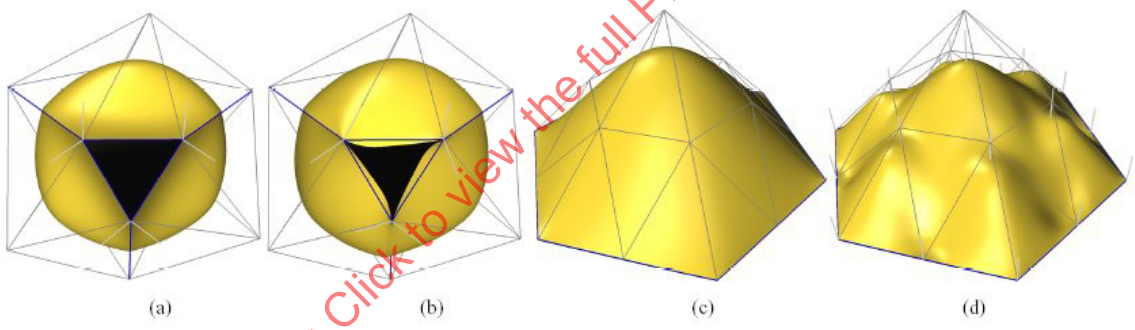


Figure 15 — Normal interpolation. (a) Surface with convex corners. (b) Prescribed directions: at each corner we tilt the normal for one surface sector slightly inwards. (c) Smooth surface. (d) Same control mesh but all normals vertical.

3.3.2.3 SubdivSurfaceSector

3.3.2.3.1 Node interface

SubdivSurfaceSector { #%NDT=SFSubdivSurfaceSectorNode

field	SFInt32	faceIndex	0	# face index from coordIndex
field	SFInt32	vertexIndex	0	# vertex index from coord
exposedField	SFInt32	tag	0	
exposedField	SFFloat	flatness	0	
exposedField	SFFloat	theta	0	
exposedField	SFVec3f	normal	0 0 0	
exposedField	SFFloat	normalTension	0	

}

3.3.2.3.2 Functionality and semantics

The **faceIndex** field contains the index of the face and the **vertexIndex** field contains the vertex index the sector is attached to. The indices refer to point index and face index from **coord** and **coordIndex** fields respectively in the parent **SubdivisionSurface** node.

The **tag** field indicates the type of sector (Figure 16):

Tag value	Semantic
0	No corner sector
1	Convex sector
2	Concave sector

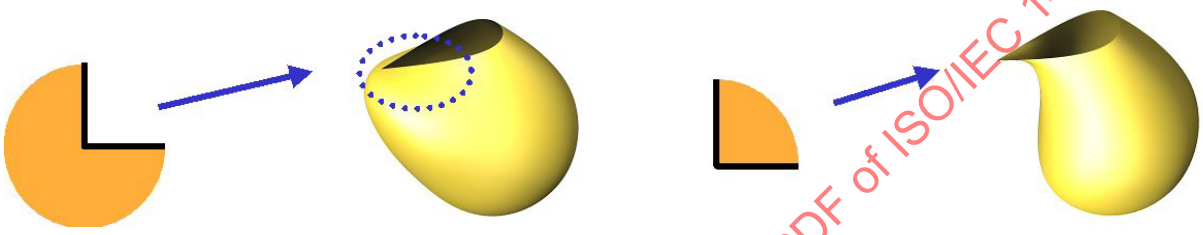


Figure 16 — Concave corners (left) and convex corners (right).

The **flatness** field contains a value between 0 and 1: 0 indicating no modified flatness, and 1 indicating maximal flatness. This parameter is denoted *s* in Loop and Catmull-Clark rule definitions above (Figure 17).

Figure 17 — Influence of the flatness parameter.

The **theta** field is an angle used to control the shape of the corner; this angle is denoted θ_k in the mask definitions above (Figure 18).

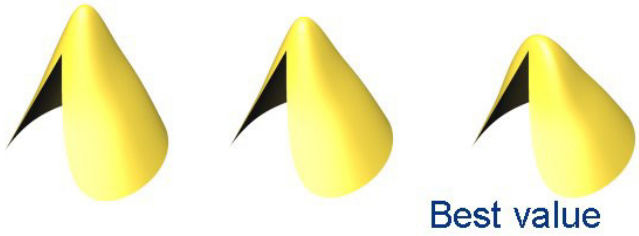


Figure 18 — Influence of theta.

The **normal** field specifies the desired normal at the vertex (Figure 19).

The **normalTension** field is the influence of the normal between 0 and 1, 0 meaning no influence; denoted *t* in the formulas for normal modification (Figure 19).

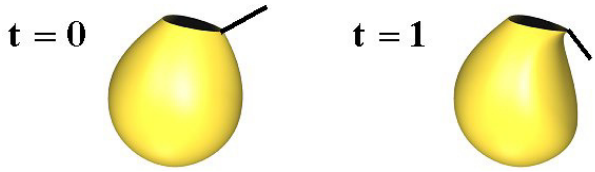


Figure 19 — Prescribed normals and tension parameter.

3.3.2.4 WaveletSubdivisionSurface

3.3.2.4.1 Node interface

```
WaveletSubdivisionSurface { #NDT=%SF3DNode
    exposedField    SFGeometryNode      baseMesh        NULL
    exposedField    SFFloat             frequency       1.0
    exposedField    SFFloat             fieldOfView     0.785398    # pi/4
    exposedField    SFInt32             quality         1
}
```

3.3.2.4.2 Functionality and semantics

Wavelet representation of surfaces relies on an extension of classical multiresolution analysis theory in which operators similar to those of subdivision surfaces are used in place of traditional interval subdivision and low-pass filtering (scaling spaces).

In this representation, a 3D mesh is received as a coarse mesh and independent sets of zero-tree encoded wavelet coefficients, allowing adaptive and progressive reconstruction at the client side.

The **WaveletSubdivisionSurface** node is used to represent adaptively multi-resolution polygonal models. It allows representing at any moment a 3D-model at a resolution that depends on the distance of the object with respect to the viewpoint. Transmission of such a model consists of two parts: a base mesh, which describes the model at the coarsest level of detail, and series of refinements (wavelet coefficients) that can be carried on one or more elementary streams (see the bitstream definition in subclause 4.3.1.1).

The **baseMesh** field shall specify either a **SubdivisionSurface** node or an **IndexedFaceSet**. If **baseMesh** is a **SubdivisionSurface**, the stream containing the mesh shall consist of a single access unit, and it will be explicitly instantiated as a **SubdivisionSurface**. The structure of this **SubdivisionSurface** will be preserved, namely deletion and insertion of new faces and/or vertices is explicitly forbidden. However, replacing the values of the coordinates is allowed. If **baseMesh** is an **IndexedFaceSet**, it is required to be manifold.

NOTE: Instantiating explicitly **baseMesh** as a **SubdivisionSurface** or **IndexedFaceSet** allows animating the node.

The terminal is responsible for maintaining an internal representation of the mesh to be rendered consisting of the **baseMesh** to which the **downstream** data must be applied.

The **frequency** field gives a preferred frequency at which the terminal shall give its state by the means of the back channel stream if it exists.

The **quality** field allows parameterizing the quality of the rendered object. The range of values for this field is [0…2] where 0 stands for low quality, 1 for normal quality, 2 for best quality. Those values don't have a more explicit semantic, that is, it is a hint for the terminal.

The player sees the **WaveletSubdivisionSurface** node as a **SubdivisionSurface** or **IndexedFaceSet** node container. That is, the player has access to the **baseMesh** field.

3.3.3 MeshGrid representation

3.3.3.1 Introduction

The peculiarity of the MeshGrid representation [68] is that it defines the object's wireframe (see e.g. Figure 20(a)) by (1) describing the connectivity between the vertices located on the surface of the object by a so-called connectivity-wireframe (see Figure 20(b)), and (2) positioning these vertices in relation to a regular 3-D grid of points, i.e. the reference-grid (see Figure 20(c)).

Both the connectivity-wireframe and the reference-grid can be represented hierarchically, as shown in Figure 20(d) and Figure 20(e) respectively for 3 levels in the hierarchy.

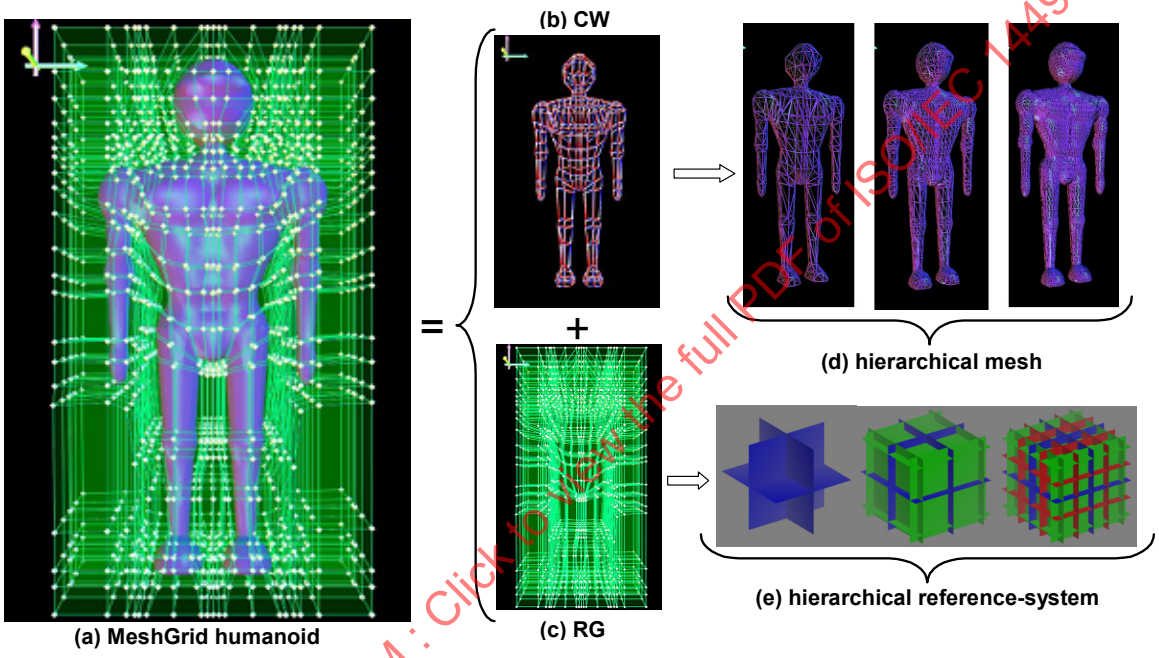


Figure 20 — A MeshGrid humanoid model designed by means of implicit surfaces.

3.3.3.1.1 The reference-grid (RG) – the reference-system of the MeshGrid model.

The reference-grid (see Figure 20(c)) is the reference system upon which the MeshGrid model is built, and is defined by the intersection points between three sets of reference-surfaces S_U, S_V, S_W as given by following equation

$$RG = \bigcap \left\{ \sum_U S_U, \sum_V S_V, \sum_W S_W \right\}$$

The discrete position (u,v,w) of a reference-grid point represents the indices of the reference-surfaces $\{S_U, S_V, S_W\}$ intersecting in that point, while the coordinate (x,y,z) of a reference-grid point is equal to the coordinate of the computed intersection point.

A multi-resolution model is designed by choosing a hierarchical reference-system, the reference-surfaces belonging to the same resolution level being displayed in the same color in the example from Figure 20(e).

The reference system should be chosen according to the topology of the object, i.e. it has higher density in (1) areas where the curvature is high, or (2) places that are deformed during the animation. For the humanoid model of Figure 20, the reference system is adjusted for traversing the anatomical articulations (joints) of the body, in order to be able to virtually split the resulting seamless connectivity-wireframe into meaningful anatomical parts (such as the shoulder, elbow, wrist).

3.3.3.1.2 The connectivity-wireframe (CW) – the surface mesh.

The connectivity-wireframe (see Figure 20(b)) is generated by merging the contouring of the original 3D model in each of the reference-surfaces S_U, S_V, S_W as given by the following equation

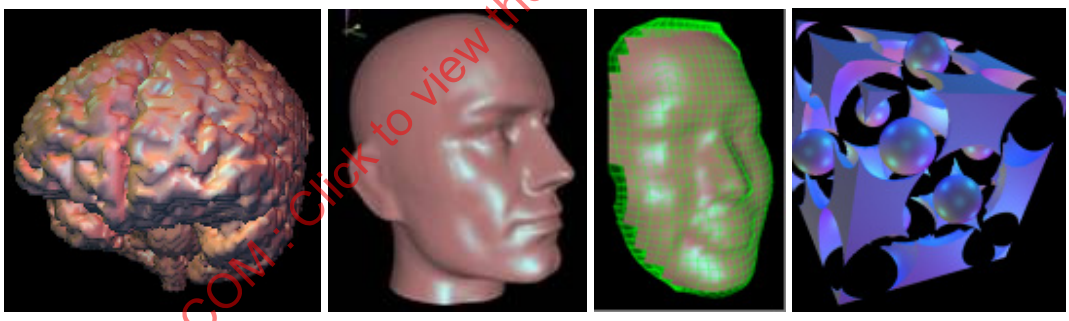
$$CW = \bigcup \left\{ \sum_U C(S_U), \sum_V C(S_V), \sum_W C(S_W) \right\}$$

in which $C(S_x)$ represents the contour obtained by the intersection with surface S_x. Each vertex is located at the intersection point between two contours.

Subdividing the reference surfaces hierarchically, as explained in subclause 3.3.3.1.1, results in a multi-resolution connectivity-wireframe, in which each resolution level consists of a single seamless mesh, as shown in Figure 20(d).

3.3.3.1.3 The features

This type of representation is specific for describing objects defined as series of contours or slices, such as: discrete 3D data sets (e.g. 3D medical images, processed range scanner data), cylindrical and spherical projected scanner meshes based on quadrilaterals, or generic models (see Figure 21(a)-(d)).



(a) (b) (c) (d)

Figure 21 — The rendered surface obtained from (a) a 3D voxel model of the brain, (b) a structured light scanner quadrilateral mesh, respectively with its deformed reference-grid in overlay in (c), and (d) a generic model.

A MeshGrid mesh allows view-dependent streaming (see subclause 4.2.3) and particular animation possibilities, such as: hierarchical grid animation, and offset based animation (see subclause 3.3.3.3).

3.3.3.2 MeshGrid

3.3.3.2.1 Node specification

MeshGrid {

eventIn	MFInt32	set_colorIndex	
eventIn	MFInt32	set_coordIndex	
eventIn	MFInt32	set_normalIndex	
eventIn	MFInt32	set_texCoordIndex	
exposedField	SFColorNode	color	NULL
exposedField	SFCoordinateNode	coord	NULL

exposedField	SFInt32	displayLevel	0
exposedField	SFInt32	filterType	0
exposedField	SFCoordinateNode	gridCoord	NULL
exposedField	SFInt32	hierarchicalLevel	0
exposedField	MFInt32	nLevels	[]
exposedField	SFNormalNode	normal	NULL
exposedField	MFInt32	nSlices	[]
exposedField	SFTextureCoordinateNode	texCoord	NULL
exposedField	MFInt32	vertexLink	[]
exposedField	MFFloat	vertexOffset	[]
field	MFInt32	colorIndex	[]
field	MFInt32	coordIndex	[]
field	MFInt32	normalIndex	[]
field	SFBool	solid	TRUE
field	MFInt32	texCoordIndex	[]
eventOut	SFBool	isLoading	
eventOut	MFInt32	nVertices	

}

3.3.3.2.2 Node semantics

The **MeshGrid** node is derived from the existing **IndexedFaceSet** node. The following fields have been removed, since the triangulation of the surface is pre-defined in the decoder: *ccw*, *convex*, *creaseAngle*. Some new fields are introduced, specific to the MeshGrid representation. The mesh is manifold, but can be open or closed.

The fields of the MeshGrid node can be encoded either (1) by BIFS, or (2) as a binary stream. When encoded as a binary stream, the following fields of the MeshGrid node will be initialized when decoding the binary stream (see subclause 4.2.3.2.36): **nSlices**, **nLevels**, **gridCoord**, **coord**, **coordIndex**, **vertexOffset**, **vertexLink**, **nVertices**. The **isLoading** flag is set to true while decoding the binary stream, respectively becomes false when the decoding finishes. The value of **isLoading** flag has to be checked before updating the fields in order to make sure that the decoder has finished modifying them. If encoded by BIFS only the **vertexLink** (the description of the connectivity-wireframe) and **vertexOffset** (the vertices' offsets relative to the reference-grid) fields need to be decoded, as explained in subclauses 4.2.3.3.1 and 4.2.3.3.3.

The **nSlices** field specifies the number of slices of the three sets $\{S_U,S_V,S_W\}$ of reference-surfaces defining the reference-grid at the last resolution level. The number of slices in the *U* direction equals to **nSlices[0]**, respectively the number of slices in the *V* direction equals to **nSlices[1]**, and the number of slices in the *W* direction equals to **nSlices[2]**. The minimum number of slices in the *U* and *V* directions is "2" while in the *W* direction the minimum number of slices is "1", in which case the reference-grid is single layer. More detail can be found in subclauses 4.2.3.3.4 and B.1.2.

The **nLevels** field defines the number of resolution levels of the MeshGrid mesh. The number of resolution levels in the *U* direction equals to **nLevels[0]**, respectively the number of resolution levels in the *V* direction equals to **nLevels[1]**, and the number of resolution levels in the *W* direction equals to **nLevels[2]**. More detail can be found in B.1.2.

The **gridCoord** field defines the reference-grid points. The number of reference-grid points can be either equal (1) to the number of reference-grid corners or (2) to the following value: $nSlices[0]\times nSlices[1]\times nSlices[2]$. In case (1), the values represent the coordinates of the reference-grid corners, and the decoder will generate a regular distributed reference-grid based on the coordinates of these corners. The number of corners can either be 1, 2, 4 or 8 depending on the type of model, i.e. single layer and cyclic, only single layer, or generic. In this case the **gridCoord** buffer is resized and its initial values are replaced by the computed coordinates of the grid points. In case (2) the values represent the coordinates of the reference-grid points. Any point from the reference-grid can be addressed by a distinct (u,v,w) discrete position, where $\{u,v,w\}$ are integer values in the range: $u \in \{0,...,nSlices[0]-1\}$, $v \in \{0,...,nSlices[1]-1\}$, and $w \in \{0,...,nSlices[2]-1\}$. Given *u* the row index, *v* the column index, and *w* the plane index, the ordering of the reference-grid points in the **gridCoord** buffer is row first, column second, and plane third. When the

MeshGrid node is initialized from a binary stream, then the contents of **gridCoord** is ignored, and it will be overridden with the coordinates decoded from the binary stream as explained in subclause 4. When the **gridCoord** field is updated by BIFS, the coordinates of the grid points that did not change can be updated in a hierarchical way as explained in subclause 3.3.3.3.1.

The **vertexLink** field defines the connectivities between the vertices in the connectivity-wireframe. The decoding of the contents of this field is explained in subclause 4.2.3.3.1. If the node is initialized from a binary stream, then the content of vertexLink is ignored.

The **vertexOffset** field specifies the scalar offset for each vertex relative to a corresponding reference-grid point. The meaning of the offset is explained in subclause 3.3.3.3.2, while the decoding of this field is described in subclause 4.2.3.3.3. The value of the scalar offset lies in the range [0, 1). If the node is initialized from a binary stream, then the content of vertexOffset is ignored, and it will be overridden with the scalar offsets decoded from the binary stream.

The **coord** field is filled with the coordinates (*x*,*y*,*z*) of the vertices, as result of the decoding procedure (see subclause 4.2.3.3.1). Its contents, if any, will always be overridden. The coordinates of the vertices can be derived from the coordinates of the reference-grid points as explained in subclause 4.2.3.3.1.

The **coordIndex** field is filled with the indices of the triangles when building the triangulation of the decoded mesh as explained in 4.2.3.3.1.3. Its contents, if any, will always be overridden.

The **nVertices** field stores the number of vertices for each resolution level. This field is updated after the decoding procedure (see subclause 4.2.3.3.1).

The **hierarchicalLevel** field defines the resolution level of the reference-grid points (**gridCoord** field) that may change during the animation. The usage is explained in subclause 4.2.3.3.1.

The **filterType** field defines the type of filter used for the hierarchical interpolation of the reference-grid points during the animation.

filterType	Meaning
00	Use short filter (see subclause 4.2.3.3.1)
01	Use smooth filter (see subclause 4.2.3.3.1)
10	Reserved
11	Reserved

The **displayLevel** field specifies the resolution level of the MeshGrid model to be rendered. Accepted values are in the range [0, maxLevel], where maxLevel is the maximum value between **nLevels[0], nLevels[1], nLevels[2].** More detail can be found in Annex B.1.2.

The fields **normal**, **normalIndex**, **texCoord**, **texCoordIndex**, **color**, **colorIndex,** and **solid** have the same semantic as for the **IndexedFaceSet** node.

3.3.3.3 Animation Extensions

3.3.3.3.1 Hierarchical interpolation of the reference-grid points

In case of a multi-resolution MeshGrid model, animating the hierarchical reference-grid can be done on a hierarchical basis, more precisely any change in the coordinates of reference-grid points, i.e. the ***gridCoord*** field (see subclauses 3.3.3.2.1 and 3.3.3.2.2), at a certain resolution level will trigger a local update of the coordinates of the reference-grid points belonging to a higher resolution level. The new positions of the

reference-grid points belonging to resolution level (l+1) can be computed from the new positions of the neighboring reference-grid points at resolution level (l) via an interpolation method based on "Dyn's four point scheme for curves" [30]. The position of a reference-grid point P_{2n}^{l+1} (shown in Figure 22) is computed as follows:

$$P_{2n}^{l+1} = \Delta P_{2n}^{l+1} + (-w \cdot P_{2n-3}^{l} + (0.5+w) \cdot P_{2n-1}^{l} + (0.5+w) \cdot P_{2n+1}^{l} - w \cdot P_{2n+3}^{l})$$

$$\Delta P_{2n}^{l+1} = P_{2n}^{l+1} + (-w \cdot P_{2n-3}^{l} + (0.5+w) \cdot P_{2n-1}^{l} + (0.5+w) \cdot P_{2n+1}^{l} - w \cdot P_{2n+3}^{l})$$

where l represents the hierarchical level of the grid point, P the last position of a point, P the new position, ΔP the detail computed for the last position P. The detail ΔP is added to the interpolated value in point P. The weight w defines the smoothness of the limit curve, and can be specified via the filterType field of the MeshGrid node. The weight w may have one of the following two values: w = 0 when filterType = 0 and w = 1/16 when filterType = 1. In general w is taken equal to 1/16 which corresponds to fitting a Catmull-Rom or Cardinal spline curve through the points.

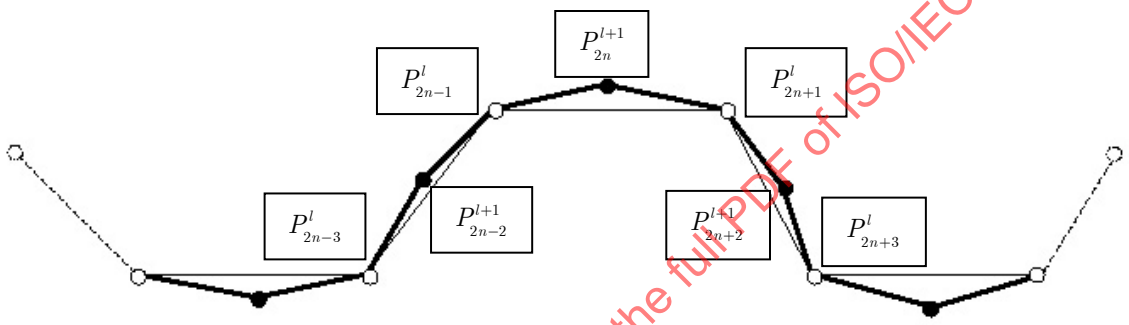


Figure 22 — Dyn's four point scheme for curves applied for the hierarchical MeshGrid.

The ***hierarchicalLevel*** field specifies the resolution level at which the grid points, i.e. the ***gridCoord*** field, may change. The interpolation of the grid points will take place only for the range of resolution levels [***hierarchicalLevel*** + 1, ***nLevels*** – 1]. If the value of ***hierarchicalLevel*** ≥ ***nLevels*** – 1 than no interpolation occurs. In the third case, when ***hierarchicalLevel*** = -1, grid points may change at any resolution level simultaneously, the points that change are detected, and the interpolation of the remaining grid points is done according to equation above.

The position of the vertices is updated from the grid positions as explained in subclause 3.3.3.3.2.

3.3.3.3.2 Relationship between the reference-grid points and vertices

The relationship between the vertices of the connectivity-wireframe and reference-grid points is explained in the 2D cross-section of Figure 23, in which the reference-grid is put on top of the object's section.

Any reference-grid line *l* (label 1) is the result of the intersection between two reference-surfaces S_1 and S_2, belonging to two different sets (see subclause 3.3.3.1.1). Every vertex *V*, lying on a contour (label 2) of the object, stores the discrete position (*u,v,w*) – which has been derived during decoding – of the grid point it is attached to. The vertices are related to the reference-grid points in two different ways, as follows:

- As shown in Figure 23(a)-(b), vertex V is positioned in-between two grid points, one inside the object G1 to which it is attached, and one outside the object G2. The relative position of vertex V with respect to G1 and G2 is given by the scalar offset (label 3) expressed by $offset = \left|\overrightarrow{G_1V}\right| \Big/ \left|\overrightarrow{G_1G_2}\right|$ with

$offset \in [0,1)$. Because offset indicates the fractional position between G1 and G2, any displacement of G1 or G2 (see the example of Figure 23(b)) is automatically reflected in an update of the coordinates (x,y,z) of V: $\overrightarrow{OV} = \overrightarrow{OG_1} + \overrightarrow{G_1G_2} \cdot offset$.

- As illustrated in Figure 23(c), vertex V is attached in this case to a grid position G belonging to a single-layer reference-grid, i.e. the number of slices is equal to 1 in one of the {u,v,w} directions. The relative position of vertex V with respect to G is given by the scalar offset (label 3) expressed by $offset = \left|\overrightarrow{G_1V}\right| / offAmp + 0.5$ with $offset \in [0,1)$, where *offAmp* is the maximum amplitude of the offset defined by the *offsetAmplitude* variable specified in MeshGridDecoderConfig (subclause 4.2.3.2.2). Similarly to the observation made at the previous point, any displacement of G is automatically reflected in an update of the coordinates (x,y,z) of V: $\overrightarrow{OV} = \overrightarrow{OG_1} + \vec{N}(offAmp(offset - 0.5))$ where $\vec{N}$ is the normal vector to the surface at vertex *V*. In this case the displacements along the normal follow both directions as shown for the vertices with labels 4 and 5.

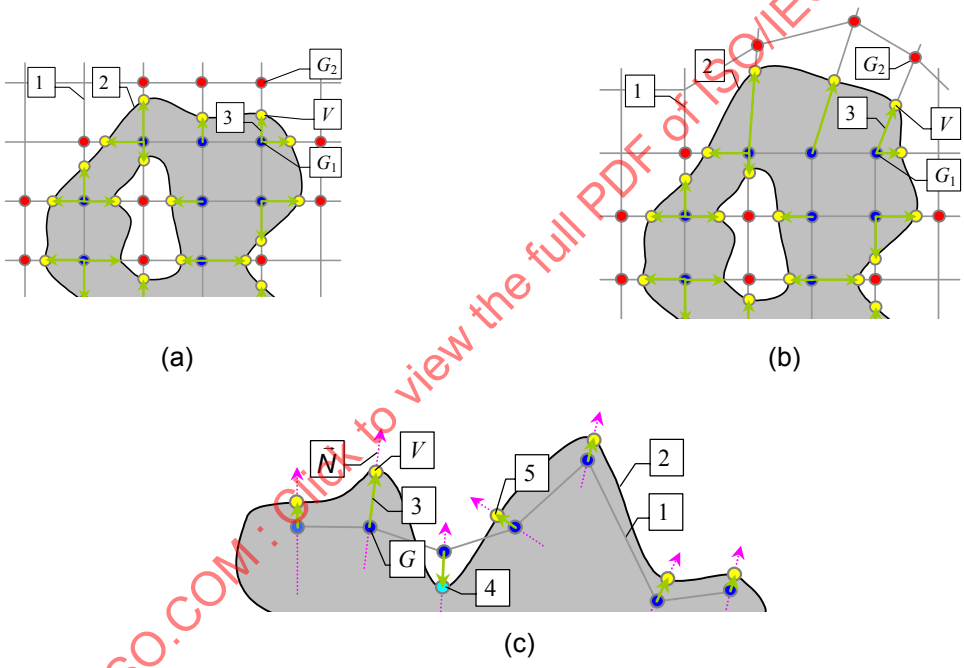


Figure 23 — A cross-section through a 3D object, illustrating the contour of the object, the reference-grid, and the relation between the vertices (belonging to the connectivity-wireframe and located at the surface of the object) and the grid points.

NOTE: In equations above O denotes the origin of the reference coordinate system in which the reference-grid is defined.

The offsets of the vertices are stored in the ***vertexOffset*** field of the **MeshGrid** node (see subclause 3.3.3.2).

3.3.3.4 Examples

The following example loads a VRML MeshGrid node.

```
Shape {
   appearance Appearance {
   material Material {
         diffuseColor 1.0 0.0 0.0
         ambientIntensity 0.1
```

```
        }
    }
    geometry MeshGrid {
      nSlices [ 3 3 3 ]
      nLevels 1
      displayLevel
      hierarchicalLevel 0
      gridCoord Coordinate {
         point [
          -1.000000 -1.000000 -1.000000,
           0.000000 -1.000000 -1.000000,
           ...
           0.000000  1.000000  1.000000,
           1.000000  1.000000  1.000000
         ]
      }
      vertexLink [
        1 1 1, 32
        2 2
        ...
        2 2
        2 2
      ]
      vertexOffset [
        0.500000
        0.200000
        ...
        0.200000
        0.500000
      ]
    }
}
```

3.3.4 MorphSpace

3.3.4.1 Introduction

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

3.3.4.2 MorphShape node

3.3.4.2.1 Syntax

MorphShape{ #%NDT=SF3DNode,SF2DNode

exposedField	SFInt32	morphID	
exposedField	SFShapeNode	baseShape	
exposedField	MFShapeNode	targetShapes	[ ]
exposedField	MFFloat	weights	[ ]

}

3.3.4.2.2 Semantics

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

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

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

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

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

with M – morphed shape, B – base shape, T_i – target shape *i*, W_i – weight of the T_i.

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

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

3.3.5 MultiResolution FootPrint-Based Representation

3.3.5.1 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.

3.3.5.2 FootPrintSetNode

3.3.5.2.1 Node Interface

```
FootPrintSetNode { #NDT=%SFGeometryNode
    exposedField MFGeometryNode children []
}
```

3.3.5.2.2 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.

3.3.5.3 FootPrintNode

3.3.5.3.1 Node Interface

```
FootPrintNode {  #NDT=%SFGeometryNode
    exposedField  SFInteger  index  -1
```

exposedField	SFIndexLineSet2D	footprint	NULL

}

3.3.5.3.2 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.

3.3.5.4 BuildingPartPrintNode

3.3.5.4.1 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	[]

}

3.3.5.4.2 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.

3.3.5.5 RoofNode

3.3.5.5.1 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

}

3.3.5.5.2 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.

3.3.5.6 FacadeNode

3.3.5.6.1 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	[]

}

3.3.5.6.2 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 regularly repeat 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.

3.3.6 Solid representation

Solid representation includes 3 geometry nodes (Implicit, Quadric and SolidRep) and extensions of the script language for implementation of the Arithmetic of Forms.

3.3.6.1 Solid modeling nodes

3.3.6.1.1 Implicit

3.3.6.1.1.1 Node interface

Implicit { # % NDT = SFGeometryNode

exposedField	SFVec3f	bboxSize	2.0 2.0 2.0	
exposedField	MFFloat	c	[ ]	# 4 coeffs for hyperplane, 10 coeffs for quadrics, 35 coeffs for quartics
exposedField	SFBool	solid	FALSE	
exposedField	SFBool	dual	FALSE	
exposedField	MFInt32	densities	[ ]	# (1, 2)

}

3.3.6.1.1.2 Functionality and semantics

The **Implicit** geometry node defines an algebraic surface represented by an implicit equation, which makes it possible to know if an unspecified point of space is inside, on or outside the volume delimited by this surface.

bboxSize: bounding box dimension.

c: coefficients of following polynomials as follows:

The implicit equation that defines a hyperplane, is as follows:

$$c_0 X_0 + c_1 X_1 + c_2 X_2 + c_3 X_3 = 0$$

The implicit equation of the second degree that defines quadrics, is as follows:

$$c_0 X_0^2 + c_1 X_0 X_1 + c_2 X_1^2 + c_3 X_0 X_2 + c_4 X_1 X_2 + c_5 X_2^2 + c_6 X_0 X_3 + c_7 X_1 X_3 + c_8 X_2 X_3 + c_9 X_3^2 = 0$$

The implicit equation of the fourth degree that defines quartics is as follows:

$$c_0 X_0^4 + c_1 X_0^3 X_1 + \ldots + c_4 X_1^4 + c_5 X_0^3 X_2 + \ldots + c_{14} X_2^4 + c_{15} X_0^3 X_3 + \ldots + c_{34} X_3^4 = 0$$

In the above equations, the point coordinates were made homogeneous by the addition of a fourth coordinate X_3.

solid indicates if the geometry is solid (TRUE) or is a surface (FALSE)

dual indicates that a unary duality operation has to be applied to volume initialization. Implies **solid** == TRUE.

densities has 2 values: the first value is the density code associated to the skin and the second value is the density associated to the inside.

The surface is only created when the node is displayed allowing its inclusion into a **SolidRep** node. The origin of the axis system is the bounding box's center. The surface will be clipped by the bounding box. If **solid** is set to TRUE, the volume will be closed and intersected with the bounding box.

If the **dual** field is TRUE, the volume is set to **solid** and the difference operation with bounding box will be carried out (bounding box – implicit solid).

The use of bounding boxes for **Implicit** nodes makes it possible to limit infinite surfaces.

The densities are used when including the primitive into a **SolidRep** node with solid operation. The density value is an Int32 and can be encoded.

As a benefit of the implicit definition, the function *isOutside (Point &p)* will return 3 values:

- 1 : if the point is outside the volume defined by the surface
- 0 : if the point is on the surface
- -1 : if the point is inside the volume defined by the surface

3.3.6.1.2 Quadric

3.3.6.1.2.1 Node interface

Quadric { #%NDT=SFGeometryNode

exposedField	SFVec3f	bboxSize	2 2 2	
exposedField	SFVec4f	P0	-1 0 0 1	
exposedField	SFVec4f	P1	1 0 0 1	
exposedField	SFVec4f	P2	0 1 0 0	
exposedField	SFVec4f	P3	0 0 1 0	
exposedField	SFVec4f	P4	0 1 0 1	
exposedField	SFVec4f	P5	0 0 1 1	
exposedField	SFBool	solid	FALSE	
exposedField	SFBool	dual	FALSE	
exposedField	MFInt32	densities	[]	# (1, 2)

}

3.3.6.1.2.2 Functionality and semantics

The **Quadric** geometry node defines a second-degree implicit surface by using 6 geometric control points.

bboxSize: bounding box dimension.

The 6 geometric control points of the quadric, in projective coordinates (see Figure 24) are:

P0, **P1:** 2 points tangent to the quadric

P2, **P3:** 2 poles of the construction tetrahedron

P4, **P5:** 2 passing points of the quadric

solid: solid (TRUE) or surface (FALSE)

dual: unary duality operation to be applied to volume initialization. Implies solid == TRUE.

densities: 2 values: the first value is the density code associated to the skin and the second value is the density associated to the inside.

Each point is defined using homogeneous coordinates allowing the point to be sent to the infinity. The values are relative to the unitary bounding box (from –1 to +1). If the absolute value of a coordinate is greater than 1, the point is outside the bounding box.

This node creates an **Implicit** node of the second degree through a geometric interface. The polynomial coefficients will be calculated according to the geometrical construction method using a construction tetrahedron and 2 passing points as described in the Quadric's construction mechanism (see subclause 3.3.6.6.1.1.2).

The quadric surface is only created when the node is displayed allowing its inclusion into a **SolidRep** node. The origin of the axis system is the bounding box's center. The surface will be clipped by the bounding boxe. If **solid** is set to TRUE, the volume will be closed and intersected with the bounding box.

If the **dual** field is TRUE, the volume is set to **solid** and the difference operation with the bounding box will be carried out (bounding box – implicit solid).

A continuous volume deformation is implemented by moving the control points.

The densities are used when including the primitive into a **SolidRep** node with solid operation (see subclause 3.3.6.1.3). The density value is encoded as a 32-bit integer.

- **Quadric's Construction Mechanism**

Based on the principle of projective geometry, a geometrical construction mechanism for quadrics has been defined in [63], [65], [52] according to Pascal's Theorem.

In fact, the geometric control of any quadric goes via the extension to 3D of the principle for constructing conics, using a construction triangle and a passing point. The quadric will be constructed and controlled with the help of a "construction tetrahedron" and two passing points located in the planes defined by two particular faces of the tetrahedron. A set of six points will thus allow us to define the quadric from two conic sections that intersect in space at the two contact points on the tangent planes.

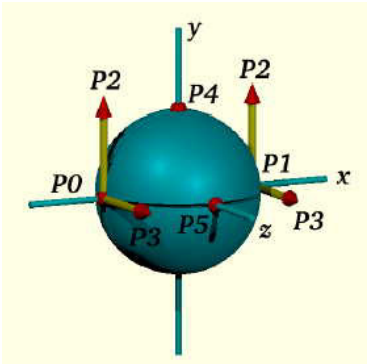


Figure 24 — Quadric's 6 geometric control points

The sphere in the above figure is constructed with the help of two circular sections belonging to two perpendicular planes. The two points P2 and P3 that serve as poles have been sent to infinity respectively along the Y-axis and the Z-axis (fourth coordinate equal to 0).

In general, for any quadric - projective, affine or metric - we shall define just six control points, P0 to P5, such that P0 and P1 are two contact points on two tangent planes to the quadric, P2 and P3 are two poles of the quadric and therefore belong to the two tangent planes, and P4 and P5 are two passing points for the quadric defining two conic sections of the quadric in the planes P0-P1-P2 and P0-P1-P3.

It is possible to determine the 10 coefficients of the quadric's general equation from the previously described quadric's geometrical construction system. (Reference: [63] – pages 100 – 114).

- **Projective Coordinates System**

In order to define all the quadrics with the 6 control points, it is imperative to remain in projective space. The coordinates system used in projective space is that of Grassmann-Plücker (one refers then to homogeneous coordinates), in which a point, a line and a plane respectively have 4, 6 and 4 coordinates. It can be interesting to make a parallel with non-homogeneous Euclidean coordinates. In this respect, one uses a representation of the point in 3 dimensions of the form (X, Y, Z). Adding to this triplet a fourth coordinate equal to 1 does not modify in any way the correct operation of all geometrical procedures known as Euclidean. The first 3 coordinates will indifferently represent integers or decimal numbers, provided that the fourth coordinate remains equal to 1. By particularizing the fourth coordinate and giving it a 0 value, one goes from a Euclidean coordinates system to an affine coordinates system, in which it becomes possible to control and use the points to infinity. Thus, an observer located in (1 1 1 0) will be positioned at infinity in the direction determined by its Euclidean position (1 1 1 1) and the origin of the coordinates system (0 0 0 1). In addition to the possibility of managing infinity in a natural way, this approach makes it possible to solve the problem of representation and calculation of rational coordinates, which is fundamental to obtain perfect precision whenever the coordinates are initially rational or brought about to become rational. Let us consider for example the homogeneous coordinates point (1/3 5/6 0 1); it will be represented by the approximate quadruplet (0.333 0.833 0 1) or, much better, by the quadruplet (2/6 5/6 0/6 6/6) also noted (2: 5: 0: 6).

3.3.6.1.3 SolidRep

3.3.6.1.3.1 Node interface

SolidRep { #%NDT=SFGeometryNode

exposedField	SFVec3f	bboxSize	2.0 2.0 2.0
exposedField	SF3DNode	solidTree	NULL
exposedField	MFInt32	densityList	[]

}

3.3.6.1.3.2 Functionality and semantics

The **SolidRep** geometry node holds the solid tree resulting of solid operations on solid primitives and/or others **SolidRep** nodes, and/or other BIFS 3D geometrical nodes.

bboxSize contains the bounding box dimensions that will be used for clipping.

solidTree contains the geometry to be initialized as solid. The following nodes may be included under a **solidTree** field: **Group**, **Transform**, **Shape**, **Implicit**, **Quadric**, **SolidRep**, **Sphere**, **Cone**, **Cylinder**, **IndexedFaceSet**, and **Box**.

The **solidTree** field can also be modified by solid operations applied by a script.

densityList: set of densities to select for display . If the list is empty, the default value will be all densities.

A **SolidRep** node can be included in a **Transform** node. The implicit operation of the children of a **Transform** node or of any grouping node is the ternary union operation.

Texture mapping onto implicit surfaces can be done using any algorithm. The gradient algorithm [90] is an example of an algorithm well suited for implicit surfaces.

3.3.6.2 Scripting extensions for solid modeling operations

The result of a solid operation is always preserved in the **solidTree** field of a **SolidRep** node. The **SolidRep** node is always the root of a solid tree. This solid tree is not processed until the display request.

Only the root of the complete tree will be processed (and not the sub-trees).

These operations are accessed through script language, whose selected operators (+, -...) were overloaded for operations between **Geometry** nodes.

A non-implicit solid geometrical node is first of all transformed into a **SolidRep** node before being used as a solid operation's operand. If this geometry does not define a closed volume, it will not be considered.

The syntax of the solid operation is as follows:

SolidRep.solidTree = {Geometry / Int32} < op > {SolidRep/Implicit/Quadric}

The operands are:

- An implicit primitive (Implicit or Quadric) with "solid = TRUE" and densities;
- Or a SolidRep containing a sub-solid tree;
- Or a non-implicit geometry (only solid primitives are supported such as Sphere, Cylinder, Cone, Box or closed shapes defined by IndexedFaceSet);
- Or an 32-bit integer (Int32) value representing the density of the entire space (used as operand for implicit filtering of densities).

An only-assignment expression makes it possible to convert a non-implicit geometrical primitive into a bound implicit solid.

The complete set of solid operations <**op**> is composed of general arithmetic operators, logic operators (ternary logic) and filtering operators. A left to right evaluation of the expression is applied. The parenthesis can be used for grouping with precedence on the left to right evaluation.

3.3.6.2.1 General arithmetic operators

The operators below achieve combinations of solid forms by isolating or selecting certain regions in a combinatorial way by means of their densities. The densities are always positive integer.

Table 2 — General arithmetic operators

Op	Description	Rules for each P(X1,X2,X3,X4)	Syntax
Sadd	Arithmetic addition of the density of two forms	$d_r(P) = d_0(P) + d_1(P)$	**Sadd (F0, F1)**
Smul	Arithmetic multiplication of the density of two forms	$d_r(P) = d_0(P) * d_1(P)$	**Smul (F0, F1)**
Sdif	The positive difference of two forms F0 and F1.	Returns the difference between the densities if this result is nonnegative, and 0 if the result is negative.	**Sdif (F0, F1)**
Sexp	Exponentiation of forms	Raises one form to the power of another form. The density of a point with respect to F1 serves as exponent to the density of the same point	**Sexp (F0, F1)**

		with respect to F0.	
Sgcd	Greatest common divisor	Returns the gcd of the densities of two forms.	**Sgcd (F0, F1)**
Slcm	Least common multiple	Returns the lcm of the densities of two forms.	**Slcm (F0, F1)**
Smod	Integral remainder	The Integral remainder operator calculates the integer remainder when the density of a point with respect to F0 is divided by the density of that point with respect to F1	**Smod (F0, F1)**
Ssab	Absolute difference	Returns the result of the subtraction between the densities of two forms when this result is nonnegative. Otherwise, the return is the result opposed value.	**Ssab (F0, F1)**
Scub	Integral cube root	Returns the integral cube root of the density of the form.	**Scub (F0)**
Ssqr	Integral square root	Returns the integral square root of the density of the form.	**Ssqr (F0)**
Smax	Maximum	This operator is equivalent to the ternary union for n-ary logic	**Smax (F0, F1)**
Smin	Minimum	This operator is equivalent to the ternary intersection for n-ary logic.	**Smin (F0, F1)**

3.3.6.2.2 Logic operators (Ternary logic)

The forms are coded with ternary logic: the density 0 for the outside, the density 1 for the skin and the density 2 for the inside.

Note: if used on an n-ary logic, the densities are taken modulo 2 to convert the value into the ternary logic.

Table 3 — Ternary logic operators

<table>
<tr><th>Op</th><th>Description</th><th>Rules for each P(X1,X2,X3,X4)</th><th>Syntax</th></tr>
<tr><td>Suni</td><td>Ternary union of two forms.</td><td>
F1
<table>
<tr><td>Suni</td><td>0</td><td>1</td><td>2</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>2</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>2</td></tr>
<tr><td>2</td><td>2</td><td>2</td><td>2</td></tr>
</table>
F0
</td><td>Suni (F0, F1)</td></tr>
<tr><td>Sint</td><td>Ternary intersection</td><td>
F1
<table>
<tr><td>Sint</td><td>0</td><td>1</td><td>2</td></tr>
</table>
</td><td>Sint (F0, F1)</td></tr>
</table>

		F0 0 0 0 0 1 0 1 1 2 0 1 2	
Simp	Ternary implication	**F1** **F0** **Simp** 0 1 2 0 2 2 2 1 1 1 2 2 0 1 2	**Simp (F0, F1)**
Simr	Reciprocal Ternary implication	**F1** **F0** **Simr** 0 1 2 0 2 1 0 1 1 1 1 2 0 1 2	**Simr (F0, F1)**
Sdua	**Ternary dual of the volume**	**F0** **Sdua** 0 2 1 1 2 0	**Sdua (F0)**

3.3.6.2.3 Filtering/test operators

In addition a set of test functions can be applied on ***SolidRep*** nodes. These functions are used to filter densities while keeping the filtering inside the solid tree instead of defining the densities to be considered during display time.

Table 4 — Filtering and test operators

Filter	Description	Rules	Syntax
Seqf	Equality filter	The density of the volume that checks the equality test, and 0 otherwise.	**Seqf (F0, F1)**
Sgtef	greater than or equal filter	The density of the second volume if the density of the first one is greater than or equal to the density of the second volume, and 0 otherwise.	**Sgtef (F0, F1)**
Sgtf	Greater than	The density of the second volume if the density of the first one is greater than the density of the	**Sgtf (F0, F1)**

	filter	second volume, and 0 otherwise.	
Sltef	Less than or equal filter	The density of the second volume if the density of the first one is less than or equal to the density of the second volume, and 0 otherwise.	**Sltef (F0, F1)**
Sltf	Less than filter	The density of the second volume if the density of the first one is less than the density of the second volume, and 0 otherwise.	**Sltf (F0, F1)**
Sevnf	Even filter	The volume density if the density is even, and 0 otherwise	**Sevnf (F0)**
Soddf	Odd filter	The result is the volume density if the density is odd, and 0 otherwise.	**Soddf (F0)**
Sneqf	Difference filter	The result is the second volume density if the densities are different, and 0 otherwise.	**Sneqf (F0, F1)**

Scripts are used through Script node. A Script node can be included as descending from any grouping node but is independent of the current coordinate system.

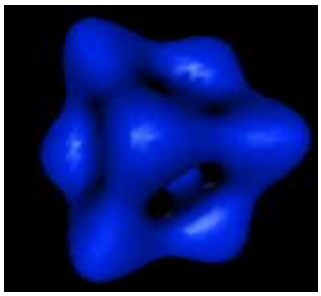
3.3.6.3 Examples

3.3.6.3.1 A cell primitive

Dual of quarter cylinder defined by projective hexahedron

```
# Dual of quarter cylinder
PROTO GDlychxa [
   fieldSFVec3f size 5 5 5
   field SFBool dual TRUE
   ]{
      Quadric {
   P0 -3 0 0 1
   P1  1 0 1 1
   P2  0 1 0 0
   P3  0 0 1 0
   P4  0 1 0 0
   P5  -1 0 1 1
            bboxSize IS size
   dual IS dual
     }
}
```

3.3.6.3.2 A quartic defined by 35 coefficients



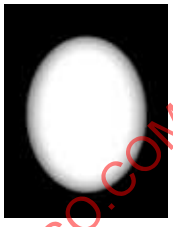
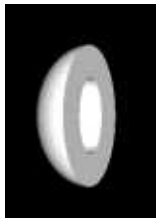
Tangle cube

VRML code:

```
Group {
   children Shape {
# tangle cube
      geometry Implicit {
         bboxSize 5 5 5
         solid FALSE
         c [
            -1,0,0,0,-1,0,0,0,0,0,0,0,0,0,-1,0,
            0,0,0,0,0,0,0,0,0,5,0,5,0,0,5,0,
            0,0,-10.2]
      }
   }
```

3.3.6.3.3 Solid operations and densities

Example of density use: a single model carries the characteristic of the matter and can be displayed differently

Egg white : density 1 Egg yolk : density 3 Operation: addition					
	Full egg with yolk inside	Cut by a box (density 4)	Cut by a box (density 1)	Cut by a box (density 4)	Cut by a box (density 4)
Densities displayed	1,3,4	1,4	1,3,4	1	4

VRML code:

```
# b: egg cut by a box
Group{
  children [
    # Primary SolidRep
    DEF Le_Solid3 SolidRep{
      bboxSize 5 5 5
      densityList [1 3 4] # choice of densities to display
      dual FALSE
    }
    DEF Le_Script Script{ # Solid Tree definition
```

```
      field SFNode Srepout USE Le_Solid3
      #primitives
      field SFNode White # container
      Transform {
        translation 0 0 0
        children [
          Shape {
        …
            geometry Quadric {
              P0 1 0 0 1
              P1 -1 0 0 1
              P2 0 1 0 0
              P3 0 0 1 0
              P4 0 1 0 1
              P5 0 0 1 1
              boxSize 1 1.3 0.95
              solid FALSE
            }
          }
        ]
      }
      field SFNode Yolk # matter inside
      Transform {
        children [
          Shape {
            …
            geometry Quadric {
              P0 1 0 0 1
              P1 -1 0 0 1
              P2 0 1 0 0
              P3 0 0 1 0
              P4 0 1 0 1
              P5 0 0 1 1
              bboxSize 0.7 0.7 0.4
              density 3
              solid FALSE
            }
          }
        ]
      }
      field SFNode CuttingBox # cutting tool
      Transform {
        translation 0.5 0 0
        children [
          Quadric {
            P0 -1 0 0 1
            P1 1 0 0 1
            P2 0 1 0 0
            P3 0 0 1 0
            P4 0 1 0 0
            P5 0 0 1 0
            bboxSize 1 1 1
            density 4
            solid FALSE
          }
        ]
      }
      directOutput TRUE
      url "vrmlscript:
        function initialize(){
          Srepout.solidTree = Sdif(Sadd(White,Yolk), CuttingBox);
        }
      "
    }
  ]
}
```

3.4 Texture tools

3.4.1 Depth Image-Based Representation

3.4.1.1 DepthImage

3.4.1.1.1 Node interface

DepthImage { #%NDT=SF3DNode			
exposedField	SFVec3f	position	0 0 10
exposedField	SFRotation	orientation	0 0 1 0
exposedField	SFVec2f	fieldOfView	PI/4 PI/4
exposedField	SFFloat	nearPlane	10
exposedField	SFFloat	farPlane	100
exposedField	SFBool	orthographic	TRUE
field	SFDepthTextureNode	diTexture	NULL
}			

3.4.1.1.2 Functionality and semantics

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

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

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

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

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

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

The **position, orientation, fieldOfView, nearPlane, farPlane,** and **orthographic** fields are exposedField types, which are for extrinsic parameters. The DepthImage node supports the camera movement and the changeable view frustum corresponding to movement or deformation of a DIBR object. And reference images that are suitable to the characteristic of a DIBR model are obtained in the modeling stage. Therefore, the fields that reflect the camera movement and the the changeable view frustum and the reference images in the modeling stage are used to create a view frustum and a DIBR object in the rendering stage.

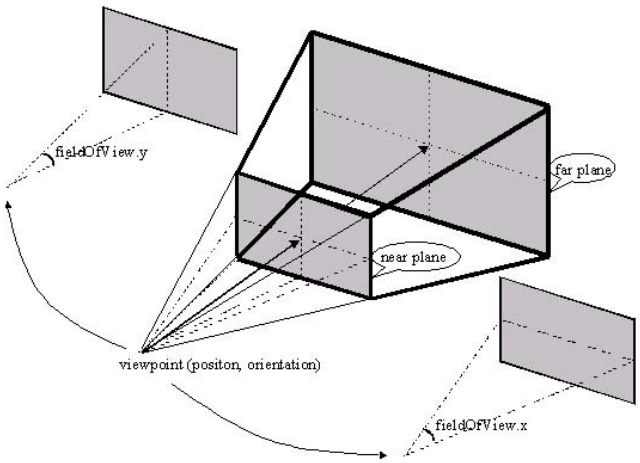


Figure 25 — Perspective view of the DepthImage

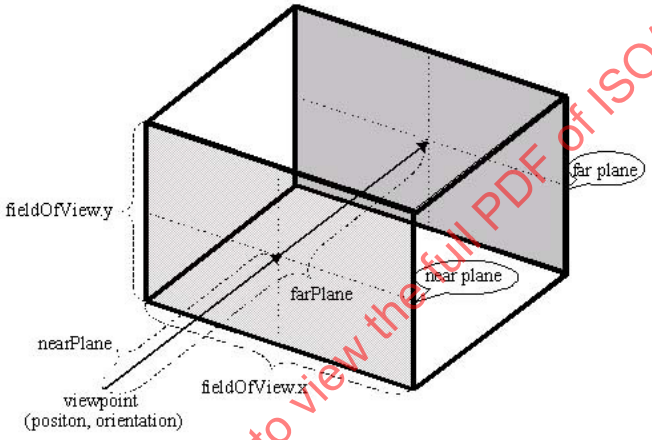


Figure 26 — Orthographic view of the DepthImage

3.4.1.2 SimpleTexture

3.4.1.2.1 Node interface

```
SimpleTexture { #%NDT=SFDepthTextureNode
    field          SFTextureNode    texture        NULL
    field          SFTextureNode    depth          NULL
}
```

3.4.1.2.2 Functionality and semantics

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

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

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

Depth field allows to compute the actual distance of the 3D points of the model to the plane which passes through the

viewpoint and parallel to the near plane and far plane:

$$dist = nearPlane + (1 - \frac{d-1}{d_{\max}-1})(farPlane - nearPlane),$$

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

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

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

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

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

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

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

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

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

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

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

3.4.1.3 PointTexture

3.4.1.3.1 Node interface

PointTexture { #%NDT=SFDepthTextureNode

field	SFInt32	width	256
field	SFInt32	height	256
field	MFInt32	depth	[]
field	MFColor	color	[]
field	SFInt32	depthNbBits	7

}

3.4.1.3.2 Functionality and semantics

The **PointTexture** node defines a multiple layers of IBR points.

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

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

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

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

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

$$d_{\max} = 2^{(depthNbBits+1)} - 1$$

3.4.1.4 OctreeImage

3.4.1.4.1 Node interface

OctreeImage { #%NDT=SF3DNode

field	SFInt32	octreeResolution	256	#%b=[1,+I]
field	MFInt32	octree	[]	#%b=[0,255] #%q=13 8
field	MFInt32	voxelImageIndex	[]	#%q=13,8
field	MFDepthImageNode	images	[]	

}

3.4.1.4.2 Functionality and semantics

The **OctreeImage** node defines a TBVO structure, in which an octree structure, corresponding image index array, and a set of **images** exist.

The **images** field specifies a set of **DepthImage** nodes with **SimpleTexture** for **diTexture** field; **depth** field in these **SimpleTexture** nodes is not used. The **orthographic** field must be TRUE for the **DepthImage** nodes. For each of **SimpleTexture**, **texture** field stores the color information of the object, or part of the object view (for example, its cross-section by a camera plane) as obtained by the orthographic camera whose position and orientation are specified in the corresponding fields of **DepthImage**. Parts of the object corresponding to each camera are assigned at the stage of model construction. The object partitioning, using the values of **position**, **orientation**, and **texture** fields, is performed so as to minimize the number of cameras (or, equivalently, of the involved **images**), at the same time to include all the object parts potentially visible from an arbitrary chosen position. The **orientation** fields must satisfy the condition: camera view vector has only one nonzero component (i.e., is perpendicular to one of the enclosing cube faces). Also, sides of the **SimpleTexture** image must be parallel to corresponding sides of enclosing cube.

The **octree** field completely describes object geometry. Geometry is represented as a set of voxels that constitutes the given object. An octree is a tree-like data structure, in which each voxel is represented by a byte. 1 in *i*th bit of this byte means that the children voxels exist for the *i*th child of that internal voxel; while 0 means that it does not. The order of the octree internal voxels shall be the order of breadth first traversal of the octree. The order of eight children of an internal voxel is shown in Figure 27. The size of the enclosing cube of the total octree is 1×1×1, and the center of the octree cube shall be the origin (0, 0, 0) of the local coordinate system.

The **voxelImageIndex** field contains an array of image indices assigned to voxel. At the rendering stage, color attributed to an octree leaf is determined by orthographically projecting the leaf onto one of the **images** with a particular index. The indices are stored in an octree-like fashion: if a particular image can be used for all the leaves contained in a specific voxel, the voxel containing index of the image is issued into the stream; otherwise, the voxel containing a fixed 'further subdivision' code is issued, which means that image index will be specified separately for each children of the current voxel (in the same recursive fashion). If the **voxelImageIndex** is empty, then the image indices are determined during rendering stage.

The **octreeResolution** field specifies maximum allowable number of octree leaves along a side of the enclosing cube. The level of the octree can be determined from **octreeResolution** using the following equation:

$$octreeLevel = \lceil \log_2(octreeResolution) \rceil$$

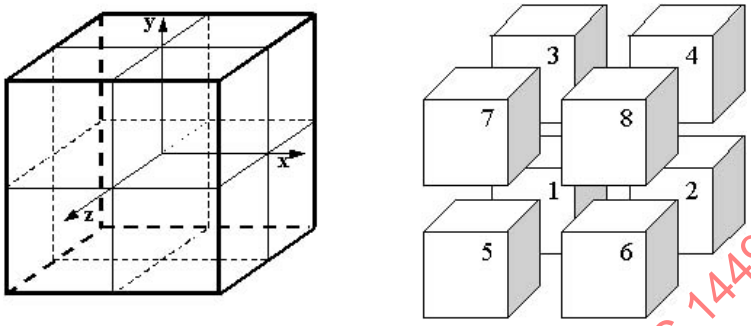


Figure 27 — The structure of octree and the order of the children

Animation of the **OctreeImage** can be performed by the same approach as the first three ways of depth-image-based animation described above, with the only difference of using octree field instead of the depth field.

3.4.2 Depth Image-based Representation Version 2

3.4.2.1 Introduction

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

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

A point-based surface consists of a set of non-uniformly distributed samples of a surface; hence we interpret it as a non-uniformly sampled signal. To continuously reconstruct this signal, we have to associate a 2D reconstruction kernel $r_k(u)$ with each sample point p_k. The kernels are defined in a local tangent frame with coordinates $u = (u, v)$ at the point p_k, as illustrated on the left in Figure 28. The tangent frame is defined by the splat and normal extensions of the DIBR structures Version 2 [94].

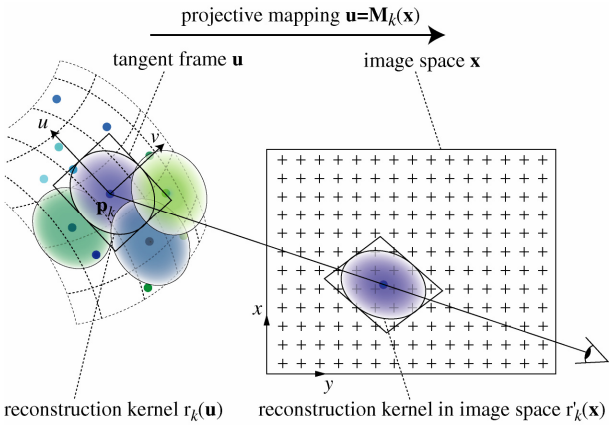


Figure 28 — Local tangent planes and reconstruction kernels

3.4.2.2 DepthImageV2 Node

3.4.2.2.1 Node interface

DepthImageV2 { #%NDT=SF3DNode

exposedField	SFVec3f	position	0 0 10
exposedField	SFRotation	orientation	0 0 1 0
exposedField	SFVec2f	fieldOfView	π/4 π/4
exposedField	SFFloat	nearPlane	10
exposedField	SFFloat	farPlane	100
field	SFVec2f	splatMinMax	0.1115 0.9875
exposedField	SFBool	orthographic	TRUE
field	SFDepthTextureNode	diTexture	NULL

}

3.4.2.2.2 Functionality and semantics

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

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

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

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

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

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

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

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

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

3.4.2.3 SimpleTextureV2 node

3.4.2.3.1 Node interface

SimpleTextureV2 { #%NDT=SFDepthTextureNode

field	SFTextureNode	texture	NULL
field	SFTextureNode	depth	NULL
field	SFTextureNode	normal	NULL
field	SFTextureNode	splatU	NULL
field	SFTextureNode	splatV	NULL

}

3.4.2.3.2 Functionality and semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.2.4 PointTextureV2 node

3.4.2.4.1 Node interface

PointTextureV2 { #%NDT=SFDepthTextureNode

field	SFInt32	width	256
field	SFInt32	height	256
field	SFInt32	depthNbBits	7
field	MFInt32	depth	[]
field	MFColor	color	[]
field	SFNormalNode	normal	
field	MFVec3f	splatU	[]
field	MFVec3f	splatV	[]

}

3.4.2.4.2 Functionality and semantics

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

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

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

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

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

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

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

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

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

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

3.4.3 Multitexturing

3.4.3.1 MultiTexture Node

3.4.3.1.1 Node Interface

MultiTexture { #%NDT=SFTextureNode

exposedField	SFFloat	alpha	1	#[0,1]
exposedField	SFColor	color	1 1 1	#[0,1]
exposedField	MFInt	function	[]	
exposedField	MFInt	mode	[]	
exposedField	MFInt	source	[]	
exposedField	MFTextureNode	texture	[]	
exposedField	MFVec3f	cameraVector	[]	
exposedField	SFBool	transparent	FALSE	

}

3.4.3.1.2 Functionality and semantics

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

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

The **cameraVector** field contains a list of camera vectors in 3D for each texture in the **texture** field. These vectors point from each associated camera to the 3D scene center. The view vectors are used to calculate texture weights according to the unstructured lumigraph approach from [82] for each render cycle, to weight all textures according to the actual scene viewpoint.

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

The **mode** field controls the type of blending operation. The available modes include MODULATE for a lit Appearance, REPLACE for an unlit Appearance and several variations of the two. However, for view-dependent Multitexturing the default mode MODULATE shall be used in conjuction with the **source** field value "FACTOR". Table 5 lists possible multitexture modes.

Table 5 — Multitexture modes

VALUE	MODE	Description
00000	MODULATE	Multiply texture color with current color Arg1 × Arg2
00001	REPLACE	Replace current color Arg2
00010	MODULATE2X	Multiply the components of the arguments, and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening.
00011	MODULATE4X	Multiply the components of the arguments, and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening.
00100	ADD	Add the components of the arguments Arg1 + Arg2
00101	ADDSIGNED	Add the components of the arguments with a -0.5 bias, making the effective range of values from −0.5 through 0.5.
00110	ADDSIGNED2X	Add the components of the arguments with a -0.5 bias, and shift the products to the left 1 bit.
00111	SUBTRACT	Subtract the components of the second argument from those of the first argument. Arg1 − Arg2

VALUE	MODE	Description
01000	ADDSMOOTH	Add the first and second arguments, then subtract their product from the sum. Arg1 + Arg2 − Arg1 × Arg2 = Arg1 + (1 − Arg1) × Arg2
01001	BLENDDIFFUSEALPHA	Linearly blend this texture stage, using the interpolated alpha from each vertex. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01010	BLENDTEXTUREALPHA	Linearly blend this texture stage, using the alpha from this stage's texture. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01011	BLENDFACTORALPHA	Linearly blend this texture stage, using the alpha factor from the MultiTexture node. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01100	BLENDCURRENTALPHA	Linearly blend this texture stage, using the alpha taken from the previous texture stage. Arg1 × (Alpha) + Arg2 × (1 − Alpha)
01101	MODULATEALPHA_ADDCOLOR	Modulate the color of the second argument, using the alpha of the first argument; then add the result to argument one. Arg1.RGB + Arg1.A × Arg2.RGB
01110	MODULATEINVALPHA_ADDCOLOR	Similar to MODULATEALPHA_ADDCOLOR, but use the inverse of the alpha of the first argument. (1 − Arg1.A) × Arg2.RGB + Arg1.RGB
01111	MODULATEINVCOLOR_ADDALPHA	Similar to MODULATECOLOR_ADDALPHA, but use the inverse of the color of the first argument. (1 − Arg1.RGB) × Arg2.RGB + Arg1.A
10000	OFF	Turn off the texture unit
10001	SELECTARG1	Use color argument 1 Arg1
10010	SELECTARG2	Use color argument 1 Arg2

VALUE	MODE	Description
10011	DOTPRODUCT3	Modulate the components of each argument (as signed components), add their products, then replicate the sum to all color channels, including alpha. This can do either diffuse or specular bump mapping with correct input. Performs the function (Arg1.R × Arg2.R + Arg1.G × Arg2.G + Arg1.B × Arg2.B) where each component has been scaled and offset to make it signed. The result is replicated into all four (including alpha) channels.
10100 – 11111		Reserved for future use

The **source** field determines the color source for the second argument. Table 6 lists valid values for the source field. For view-dependent Multitexturing “FACTOR” shall be used in conjuction with the **mode** field value MODULATE.

Table 6 — Values for the source field

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

The **function** field defines an optional function to be applied to the argument after the mode has been evaluated. Table 7 lists valid values for the **function** field.

Table 7 — Values for the function field

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

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

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

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

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

3.4.3.2 MultiTextureCoordinate Node

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

3.4.3.2.1 Node interface

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

3.4.3.2.2 Functionality and semantics

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

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

Example:

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

3.5 Animation tools

3.5.1 Deformation tools

3.5.1.1 NonLinearDeformer

3.5.1.1.1 Node interface

```
NonLinearDeformer { #%NDT=SFGeometryNode
    exposedField    SFInt32          type        0
    exposedField    SFVec3f          axis        0 0 1
    exposedField    SFFloat          param       0
    exposedField    MFFloat          extend      []
    exposedField    SFGeometryNode   geometry    NULL
}
```

3.5.1.1.2 Functionality and semantics

type is the desired deformation (0: tapering, 1:twisting, 2:bending). **axis** is the axis along which the deformation is performed, **param** the parameter of the transformation, **extend** its bounds, and **geometry** the geometry node on which is the deformation is performed or another **NonLinearDeformer** node so to chain the transformations.

Table 8 — Semantic of param and extend values for each deformation type

Type		param	Extend
0	tapering	Radius	[relative position, relative radius]*
1	twisting	Angle	[Angle min, angle max]
2	bending	Curvature	[Curvature min, curvature max, y min, y max]

For tapering, **extend** consists of a serie of 2 values: the first is the position at which the radius should be. This way a profile can be defined. The relative position along the axis of the transformation in object space: 0% at the beginning, and 100% at the end. The radius is relative to the **param** and is given in percentage. **extend** is used similarly for the other transformations.

Transformations are given by [11]:

- Tapering

To taper an object long the z-axis, x- and y-axes are just scales as a function of z:

$$(X, Y, Z) = (rx, ry, z) \quad \text{and} \quad r = f(z)$$

where $f(z)$ specifies the rate of scale per unit length along the z-axis and can be a linear or nonlinear tapering profile or function.

- Twisting

To rotate an object through an angle θ about the z-axis:

$$(X, Y, Z) = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, z) \quad \text{and} \quad \theta = f(z)$$

where $f(z)$ specifies the rate of twist per unit length along the z-axis.

- Bending

A global linear bend along an axis is a composite transformation comprising a bent region and a region outside the bent region where the deformation is a rotation and a translation. Barr defines a bend region along the y-axis as: $y_{\min} \le y \le y_{\max}$. The radius of curvature of the bend is k^{-1} and the center of the bend is at $y = y_0$. The bending angle is: $\theta = k(y' - y_0)$, where

$$y' = \begin{cases} y_{\min} & \text{if} \quad y \le y_{\min} \\ y & \text{if} \quad y_{\min} \le y < y_{\max} \\ y_{\max} & \text{if} \quad y \ge y_{\max} \end{cases}$$

The deformation is given by

$$\begin{aligned} X &= x \\ Y &= \begin{cases} -\sin\theta(z - k^{-1}) + y_0 & y_{\min} \le y \le y_{\max} \\ -\sin\theta(z - k^{-1}) + y_0 + \cos\theta(y - y_{\min}) & y < y_{\min} \\ -\sin\theta(z - k^{-1}) + y_0 + \cos\theta(y - y_{\max}) & y > y_{\max} \end{cases} \\ Z &= \begin{cases} \cos\theta(z - k^{-1}) + k^{-1} & y_{\min} \le y \le y_{\max} \\ \cos\theta(z - k^{-1}) + k^{-1} + \sin\theta(y - y_{\min}) & y < y_{\min} \\ \cos\theta(z - k^{-1}) + k^{-1} + \sin\theta(y - y_{\max}) & y > y_{\max} \end{cases} \end{aligned}$$

3.5.1.2 Free-form deformations

3.5.1.2.1 Node interface

FFD { #%NDT=SF3DNode

eventIn	MF3DNode	addChildren		
eventIn	MF3DNode	removeChildren		
exposedField	MF3DNode	children	[]	
field	SFInt32	uDimension	2	# [2, 257]
field	SFInt32	vDimension	2	# [2, 257]
field	SFInt32	wDimension	2	# [2, 257]
field	MFFloat	uKnot	[]	# (-∞,∞)
field	MFFloat	vKnot	[]	# (-∞,∞)
field	MFFloat	wKnot	[]	# (-∞,∞)
field	SFInt32	uOrder	2	# [2, 33]
field	SFInt32	vOrder	2	# [2, 33]
field	SFInt32	wOrder	2	# [2, 33]
exposedField	MFVec4f	controlPoint	[]	

}

3.5.1.2.2 Functionality and semantics

The node definition is the same as for **NURBSSurface** in subclause 3.3.1 (except for the bounds and default values of the Dimenison and Order fields) and the first 3 fields are as for a **Group** node: they enable to define the scene embedded in the FFD space.

A FFD node acts only on a scene on the same level in the transform hierarchy because a FFD applies only on vertices of shapes. If an object is made of many shapes, there are nested **Transform** nodes. If we pass

solely the DEF of these nodes, then we have no notion of what the transforms applied to the nodes are. By passing the DEF of a grouping node, which encapsulates the scene to be deformed, we can effectively calculate the transformation applied on a node.

Example

```
# The control points of a FFD are animated. The FFD encloses two shapes, which are
deformed
# as the control points move.
DEF TS TimeSensor {}
DEF PI CoordinateInterpolator4D {
   key [ … ]
   keyValue [ … ]
}

DEF BoxGroup Group {
   children [ Shape { geometry Box {} } ]
}

DEF SkeletonGroup Group {
   children [
      …# describe here a full skeleton
   ]
}
DEF FFDNode FFD {
   …# specify NURBS deformation volume
   children [
      USE BoxGroup
      USE SkeletonGroup
   ]
}

ROUTE TS.fraction_changed TO PI.set_fraction
ROUTE PI.value_changed TO FFDNode.controlPoint
```

3.5.2 Generic skeleton, muscle and skin-based model definition and animation

This subclause defines a generic animation framework for models based on skeletons and muscles.

3.5.2.1 Generic skeleton, muscle and skin-based model definition

3.5.2.1.1 SBBone

3.5.2.1.1.1 Node interface

SBBone{ #%NDT=SFSBBoneNode, SF3DNode, SF2DNode

eventIn	MF3DNode	addChildren	
eventIn	MF3DNode	removeChildren	
exposedField	SFInt32	boneID	0
exposedField	MFInt32	skinCoordIndex	[ ]
exposedField	MFFloat	skinCoordWeight	[ ]
exposedField	SFVec3f	endpoint	0 0 1
exposedField	SFInt32	falloff	1
exposedField	MFFloat	sectionPosition	[ ]
exposedField	MFFloat	sectionInner	[ ]
exposedField	MFFloat	sectionOuter	[ ]
exposedField	SFInt32	rotationOrder	0
exposedField	MFNode	children	[ ]
exposedField	SFVec3f	center	0 0 0
exposedField	SFRotation	rotation	0 0 1 0
exposedField	SFVec3f	translation	0 0 0

exposedField	SFVec3f	scale	1 1 1
exposedField	SFRotation	scaleOrientation	0 0 1 0
exposedField	SFInt32	ikChainPosition	0
exposedField	MFFloat	ikYawLimit	[ ]
exposedField	MFFloat	ikPitchLimit	[ ]
exposedField	MFFloat	ikrollLimit	[ ]
exposedField	MFFloat	ikTxLimit	[ ]
exposedField	MFFloat	ikTyLimit	[ ]
exposedField	MFFloat	ikTzLimit	[ ]

}

3.5.2.1.1.2 Functionality and semantics

SBBone node specifies data related to one bone from the skeleton.

The **boneID** field is a unique identifier which allows that the bone to be addressed at animation run-time.

The **center** field specifies a translation offset from the origin of the local coordinate system.

The **translation** field specifies a translation to the bone coordinate system.

The **rotation** field specifies a rotation of the bone coordinate system.

The **scale** field specifies a non-uniform scale of the bone coordinate system. **scale** values shall be greater than zero.

The **scaleOrientation** specifies a rotation of the bone coordinate system before the scale (to specify scales in arbitrary orientations). The **scaleOrientation** applies only to the scale operation.

The possible geometric 3D transformation consists of (in order): 1) (possibly) non-uniform scale about an arbitrary point, 2) a rotation about an arbitrary point and axis and 3) a translation.

The rotationOrder field specifies the rotation order when deals with the decomposition of the rotation in respect with system coordinate axes.

Two ways of specifying the influence region of the bone are allowed:

- Per vertex definition:

The skinCoordIndex field contains a list of indices of all skin vertices affected by the current bone. Mostly, the skin influence region of bone will contain vertices from the 3D neighborhood of the bone, but special cases of influence are also accepted.

The skinCoordWeight field contains a list of weights (one per vertex listed in skinCoordIndex) that measures the contribution of the current bone to the vertex in question. The length skinCoordIndex is equal with the length of skinCoordWeight. The sum of all skinCoordWeight related to the same vertex must be 1.

- Per bone definition:

The endpoint field specifies the bone 3D end point and is used to compute the bone length.

The sectionInner field is a list of inner influence region radii for different sections.

The sectionOuter field is a list of outer influence region radii for different sections.

The sectionPosition field is a list of positions of all the sections defined by the designer.

The falloff field specifies the function between the amplitude affectedness and distance : -1 for x^3, 0 for x^2, 1 for x, 2 for $\sin(\frac{\pi}{2}x)$, 3 for $\sqrt{x}$ and 4 for $\sqrt[3]{x}$.

The two schemes can be used independently or in combination, in which case the individual vertex weights take precedence.

The ikChainPosition field specifies the position of the bone in the kinematics chain. If the bone is the root of the IK chain then ikChainPosition=1. In this case, when applying IK scheme, only the orientation of the bone is changed. If the bone is last in the kinematics chain ikChainPosition=2. In this case, the animation stream has to include the desired position of the bone (X, Y and Z world coordinates). If ikChainPosition=3 the bone belongs to the IK chain but is not the first or the last one in the chain. In this case, position and orientation of the bone are computed by the IK procedure. Finally, if the bone does not belong to any IK chain (ikChainPosition=0), it is necessary to transmit the bone local transformation in order to animate the bone. If an animation stream contains motion information about a bone which has ikChainPosition 1, this information will be ignored. If an animation stream contains motion information about a bone which has ikChainPosition 3, this means that the animation producer wants to ensure the orientation of the bone and the IK solver will use this value as a constrain.

The **ikYawLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the X axis.

The **ikPitchLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the Y axis.

The **ikRollLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the Z axis.

The **ikTxLimit** field consists in a pair of min/max values which limit the bone translation in the X direction.

The **ikTyLimit** field consists in a pair of min/max values which limit the bone translation in the Y direction.

The **ikTzLimit** field consists in a pair of min/max values which limit the bone translation in the Z direction.

The **SBBone** node is used as a building block to describe the hierarchy of the articulated model by attaching one or more child objects. The **children** field has the same semantic as used in ISO/IEC 14496-11; the absolute geometric transformation of any child of a bone is obtained through a composition with the bone-parent transformation.

3.5.2.1.2 SBSegment

3.5.2.1.2.1 Node interface

SBSegment{ #%NDT=SFSBSegmentNode, SF3DNode, SF2DNode

eventIn	MFNode	addChildren	
eventIn	MFNode	removeChildren	
exposedField	SFString	name	“”
exposedField	SFVec3f	centerOfMass	0 0 0
exposedField	MFVec3f	momentsOfInertia	[0 0 0 0 0 0 0 0 0]
exposedField	SFFloat	mass	0
exposedField	MFNode	children	[ ]

}

3.5.2.1.2.2 Functionality and semantics

The **name** field must be present, so that the **SB**Segment can be identified at runtime. Each **SB**Segment should have a DEF name that matches the **name** field for that Segment, but with a distinguishing prefix in front of it.

The **mass** field is the total mass of the segment.

The **centerOfMass** field is the location within the segment of its center of mass. Note that a zero value was chosen for the mass in order to give a clear indication that no mass value is available.

The **momentsOfInertia** field contains the moment of inertia matrix. The first three elements are the first row of the 3x3 matrix, the next three elements are the second row, and the final three elements are the third row.

The **children** field can be any object attached at this level of the skeleton, including a **SBSkinnedModel**.

An **SBSegment** node is a grouping node especially introduced to address two issues:

- The first one is to the requirement to separate different parts from the skinned model into deformation-independent parts. Between two deformation-independent parts the geometrical transformation of one of them do not imply skin deformations on the other. This is essential for run-time animation optimization. The SBSegment node may contain as a child an SBSkinnedModel node (see the SBSkinnedModel node description below). Portions of the model which are not part of the seamless mesh can be attached to the skeleton hierarchy by using an SBSegment node;

- The second deals with the requirement to attach standalone 3D objects at different parts of the skeleton hierarchy. For example, a ring can be attached to a finger; the ring geometry and attributes are defined outside of skinned model but the ring will have the same local geometrical transformation as the attached bone.

3.5.2.1.3 SBSite

3.5.2.1.3.1 Node interface

SBSite {#%NDT=SFSBSiteNode, SF3DNode, SF2DNode

eventIn	MF3DNode	addChildren		
eventIn	MF3DNode	removeChildren		
exposedField	SFVec3f	center	0 0 0	
exposedField	MF3DNode	children	[ ]	
exposedField	SFString	name	""	
exposedField	SFRotation	rotation		0 0 1 0
exposedField	SFVec3f	scale	1 1 1	
exposedField	SFRotation	scaleOrientation	0 0 1 0	
exposedField	SFVec3f	translation	0 0 0	

}

3.5.2.1.3.2 Functionality and semantics

The **center** field specifies a translation offset and can be used to compute a bone length. The **rotation** field specifies a rotation of the coordinate system of the SBSite node.

The **scale** field specifies a non-uniform scale of the SBSite node coordinate system and the **scale** values must be greater than zero.

The **scaleOrientation** specifies a rotation of the coordinate system of the SBSite node before the scale thus allowing a scale at an arbitrary orientation. The **scaleOrientation** applies only to the scale operation.

The **translation** field specifies a translation of the coordinate system of the SBSite node.

The **children** field is used to store any object that can be attached to the **SBSegment** node.

The **SBSite** node can be used for three purposes. The first is to define an "end effector", i.e. a location which can be used by an inverse kinematics system. The second is to define an attachment point for accessories such as clothing. The third is to define a location for a virtual camera in the reference frame of a **SB**Segment node.

SBSite nodes are stored within the **children** field of an **SBSegment** node. The **SB**Site node is a specialized grouping node that defines a coordinate system for nodes in its **children** field that is relative to the coordinate systems of its parent node. The reason a **SB**Site node is considered a specialized grouping node is that it can only be defined as a child of a SBSegment node.

3.5.2.1.4 SBMuscle

3.5.2.1.4.1 Node interface

SBMuscle{ #%NDT=SFSBMuscleNode, SF3DNode, SF2DNode

exposedField	MFInt32	skinCoordIndex	[ ]
exposedField	MFFloat	skinCoordWeight	[ ]
exposedField	SFNode	muscleCurve	NULL
exposedField	SFInt32	muscleID	0
exposedField	SFInt32	radius	1
exposedField	SFInt32	falloff	1

}

3.5.2.1.4.2 Functionality and semantics

The **skinCoordIndex** field consists of a list of vertex indices from the skinned model skin which are affected by the “muscle”.

The **skinCoordWeight** field consists of a list of weights indicating in what measure a vertex in affected by the “muscle”.

The **muscleCurve** field is a NurbCurve as defined in subclause 3.3.1.2.

The **radius** field specifies the maximum distance where the “muscle” will affect the skin.

The **falloff** field specifies the function between the amplitude affectedness and distance : -1 for x^3, 0 for x^2, 1 for x, 2 for $\sin(\frac{\pi}{2}x)$, 3 for $\sqrt{x}$ and 4 for $\sqrt[3]{x}$.

Performing deformation consists in affecting the form of the muscleCurve by 1) affecting the position of the control points of the curve, 2) affecting the weight of control points or/and 3) affecting the *knot* sequence. Depending on the author, the animation stream can contain one animation mechanism or a combination of 1), 2) and 3).

At the modeling stage, each affected vertex v_i from the skin is assigned a point v_i^c from the curve, as the closest point. During animation, the translation of v_i^c obtained from the update values of **controlPoint**, **weight** or/and **knot** fields, will induce a translation on v_i:

- skinCoordWeight field is specified for vertex v_i, then:

$$Tv_i = skinCoordWeight[k] * Tv_i^c,$$

where *k* is the index of vertex v_i in the model vertices index list;

- radius field is specified, then

$$Tv_i = f(\frac{radius - d(v_i, v_i^c)}{radius}) * Tv_i^c$$

with $f(\ )$ specified by the falloff field.

3.5.2.1.5 SBSkinnedModel

3.5.2.1.5.1 Node interface

SBSkinnedModel{ #%NDT=SF3DNode,SF2DNode

exposedField	SFString	name	""
exposedField	SFVec3f	center	0 0 0
exposedField	SFRotation	rotation	0 0 1 0
exposedField	SFVec3f	translation	0 0 0
exposedField	SFVec3f	scale	1 1 1
exposedField	SFRotation	scaleOrientation	0 0 1 0
exposedField	MF3DNode	skin	[ ]
exposedField	SFCoordinateNode	skinCoord	NULL
exposedField	SFNormalNode	skinNormal	NULL
exposedField	MF3DNode	skeleton	[ ]
exposedField	MFBoneNode	bones	[ ]
exposedField	MF3DNode	muscles	[ ]
exposedField	MFSegmentNode	segments	[ ]
exposedField	MFSiteNode	sites	[ ]
exposedField	SF3DNode	weighsComputationSkinCoord	NULL

}

3.5.2.1.5.2 Functionality and semantics

The **SBSkinnedModel** node is the top of the hierarchy of Skin&Bones related nodes and contains the definition parameters for the entire seamless model or of a seamless part of the model.

The **name** field specify the name of the skinned model allowing easily identification at the animation run-time.

The **center** field specifies a translation offset from the origin of the local coordinate system.

The **translation** field specifies a translation of the coordinate system.

The **rotation** field specifies a rotation of the coordinate system.

The **scale** field specifies a non-uniform scale of the coordinate system. **scale** values shall be greater than zero.

The **scaleOrientation** specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The **scaleOrientation** applies only to the scale operation.

The **skinCoord** contains the 3d coordinates of all vertices of the seamless model.

The **skin** consists of a collection of shapes that share the same **skinCoord**. This mechanism allows considering the model as a continuous mesh and, in the same time, to attach different attributes (like color, texture) to different parts of the model.

The **skeleton** field specifies the root of the bones hierarchy.

The **bones** fields consist in the lists of all bones previously defined as **SBBone** node.

The **segments** fields consist in the lists of all bones previously defined as **SBSegment** node.

The **sites** fields consist in the lists of all bones previously defined as **SBSites** node. The **muscles** fields consist in the lists of all bones previously defined as **SBMuscle** node.

The **weighsComputationSkinCoord** field describes a specific static position of the skinned model. In many cases the static position of the articulated model defined by **skinCoord** and **skin** fields is not appropriate to compute the influence region of a bone. In this case the **weighsComputationSkinCoord** field allows specifying the skinned model vertices in a more appropriate static posture. This posture will be used just during the initialization stage and ignored during the animation. All the skeleton transformations are related to the posture defined by **skinCoord** field.

3.5.2.1.6 SBVCAnimation

This node allows to group together a set of skinned models; in order to animate them the animation data is extracted from the same resource (file or strem).

3.5.2.1.6.1 Node interface

SBVCAnimation{ #%NDT=SF3DNode,SF2DNode

exposedField	MFNode	virtualCharacters	[ ]
exposedField	MFURL	url	[ ]

}

3.5.2.1.6.2 Functionality and semantics

The **SBVCAnimation** node is a grouping node, which allows to attach a list of virtual characters to an animation stream.

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

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

3.5.2.1.7 SBVCAnimationV2

3.5.2.1.7.1 Introduction

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

3.5.2.1.7.2 Syntax

SBVCAnimationV2{ #%NDT=SF3DNode,SF2DNode

exposedField	MFNode	virtualCharacters	[ ]
exposedField	MFURL	url	[ ]
exposedField	SFBool	loop	FALSE
exposedField	SFFloat	speed	1.0
exposedField	SFTime	startTime	0
exposedField	SFTime	stopTime	0

```
    eventOut        SFTime          duration_changed
    eventOut        SFBool          isActive
    exposedField    MFInt           activeUrlIndex      []
    exposedField    SFFloat         transitionTime      0
}
```

3.5.2.1.7.3 Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.2.2 Generic skeleton, muscle and skin-based model animation

Animating a 2D/3D articulated model requires knowledge of the position of each model vertex at each key-frame. Specifying such a data is an enormous and expensive task. For this reason the AFX animation system employs the bone-based modeling of articulated models which effectively attaches the model vertices to a bone hierarchy (skeleton).Moreover, a set of curve-based muscles allow to add local deformation of the skin. This technique avoids the necessity to specify the position for each vertex, only the local transformation of each bone in the skeleton and the muscle form being animated. The local bone transformation components (*translation*, *rotation*, *scale*, *scaleOrientation* and *center*) and the muscle-curve components (control points position and weights) are specified at each frame and, at the vertex level, the transformation is obtained by using the bone-vertex and muscle-vertex influence region.

To address streamed animation, the animation data is considered separately (independent of the model definition) and it will be specified for each key-frame.

Animating a skinned model is achieved through updates of the geometric transformation component of the skeleton by transforming the bones and/or to the muscle curve form.

A general transformation of a bone, as defined by the **SBBone** node, involves: translation in any direction, rotation with respect to any rotation axis, and scaling with respect to any direction. In the **SBBone** node definition the *rotation* field is defined as a SFRotation. In order to update the orientation of a bone the *rotation* field must by updated.

Within the animation resource the bone rotation is represented as a decomposition with respect with three axes. Bijectivity of the transformation between the angle-based notation and rotation matrix or quaternion representation is ensured by the *rotationOrder* field. A triplet of angles $[\theta_1, \theta_2, \theta_3]$ describes how a coordinate frame *r* rotates with respect to a static frame *s*, here, how a bone frame rotates with respect to its parent frame. The triplet is interpreted as a rotation by θ_1 around an axis A_1, then a rotation by θ_2 around an axis A_2, and finally a rotation by $\theta_3$around an axis A_3, with A_2 different from both A_1 and A_3. The axes are restricted to the coordinate axes, X, Y, and Z, giving 12 possibilities: XYZ, XYX, YZX, YZY, ZXY, ZXZ, XZY, XZX, YXZ, YXY, ZYX, ZYZ. By considering the axis either in the bone frame (*r*) or its parent frame (*s*), there are 24 possible values for *rotationOrder*.

The bone-base animation (BBA) of a skinned model is performed by frame update of the **SBBone** transformation fields (**translation**, **rotation**, **scale**, **center** and **scaleOrientation**) and/or by updating the **SBMuscle** curve control points position, control points weight or (and) knot sequence. The BBA stream contains all the animation frames or just the data at the temporal key frames. In the last case the decoder will compute the intermediate frames by temporal interpolation. Linear interpolation is used for **translation** and **scale** components and linear quaternion interpolation is used for **rotation** and **scaleOrientation** components.

Each key-frame contains two fields: an animation mask vector and an animation values vector.

The animation mask vector consists in:

- a natural integer NumberOfInterpolatedFrames which indicates to the decoder the number of frames that have to be obtained by interpolation. If zero, the decoder interprets the received frame as a normal frame and sends it to the animation engine; if not the decoder computes NumberOfInterpolatedFrames intermediate frames and sends them to the animation engine, as well as the content of the received key-frame.

- the number of bones animated in the current frame; the number of muscles animated in the current frame.

- the boneID as well as the animation mask for each animated bone. See below for the description of SBBone animation mask.

- the muscleID as well as the animation mask for each animated muscle. See below for the description of SBMuscle animation mask.

The animation values vector consists in:

- the new values for each bone transformation component which need updating.

- the new values for each muscle control point which have been translated or change the weigh and the new values of the knot sequence of NURBS curve.

A BBA file or stream can contain the information related to a maximum number of 1024 **SBBone** and 1024 **SBMuscle** nodes belonging to one or more skinned models and grouped under the same SBVCAnimation node. The identifiers fields **boneID** and **muscleID** must be unique within an SBVCAnimation node and must be in the range [0 …1023].

Two representation of the animation data are supported by the standard: uncompressed format and compressed format.

Annex E describes the uncompressed animation file format. Subclause 4.3.2 shows the syntax of the compressed animation stream.

3.5.2.3 Animation algorithm details for BBA

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 parents 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 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.

3.6 Rendering tools

3.6.1 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.

3.6.1.1 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

}

3.6.1.2 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 30. 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 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 30).

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 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 30). 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 31).

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

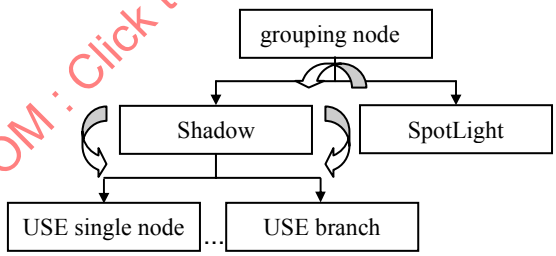


Figure 29 — Semantical representation

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

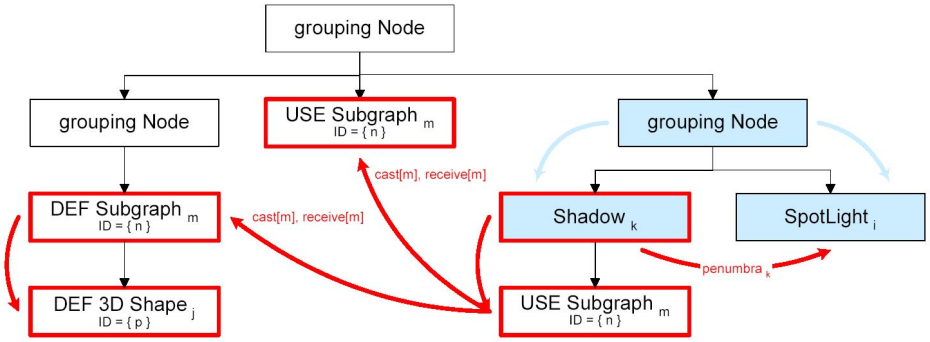


Figure 30 — Transmission of shadow properties to 3D-surfaces and light sources

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 31). Additionally it can create multiple shadow properties with a single **Shape** node.

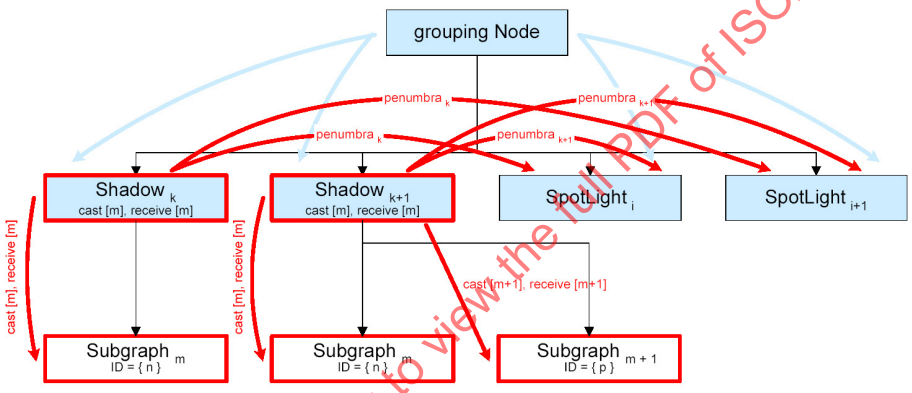


Figure 31 — Transmission of shadow properties to multiple 3D-surfaces and light sources

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.

4 AFX bitstream specification - 3D Graphics compression tools

4.1 Introduction

Clause 4 includes the definition of the bitstream syntax for compressed graphics primitives (geometry, texture, animation) and generic (multiplexing, backchannel, …) mechanisms.

The AFX compression tools can be used in different scenarios:

- connected to the BIFS nodes by using BitWrapper as defined in ISO/IEC 14496-11,
- connected to scene graph nodes defined by other standards than MPEG by following the model standardized in MPEG-4 Part 25 (ISO/IEC 14496-25),
- in a standalone file format as defined in subclause 4.5.1.

4.2 Geometry tools

4.2.1 3DMC Extension

3DMC Extension is 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.

4.2.1.1 Introduction

4.2.1.1.1 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.

4.2.1.1.2 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 32 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.*

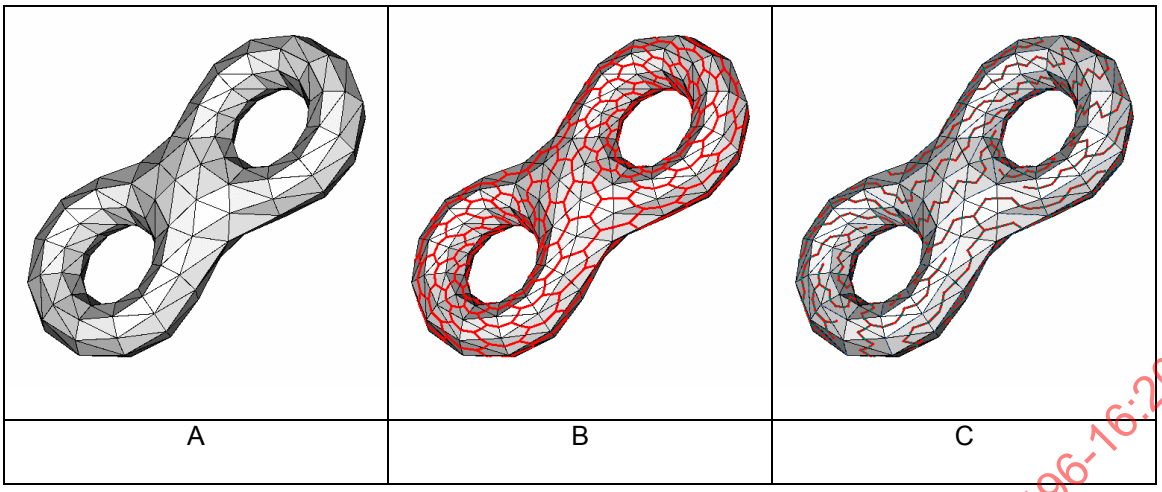


Figure 32 — A triangular 3D mesh (A), its dual graph (B), and a triangle tree (C)

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.

4.2.1.1.3 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.

4.2.1.1.4 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.

4.2.1.1.5 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.

4.2.1.1.6 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).

4.2.1.1.7 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.

4.2.1.1.8 Encoder and Decoder Block Diagrams

High level block diagrams of a general 3D polygonal model encoder and decoder are shown in Figure 33. 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.

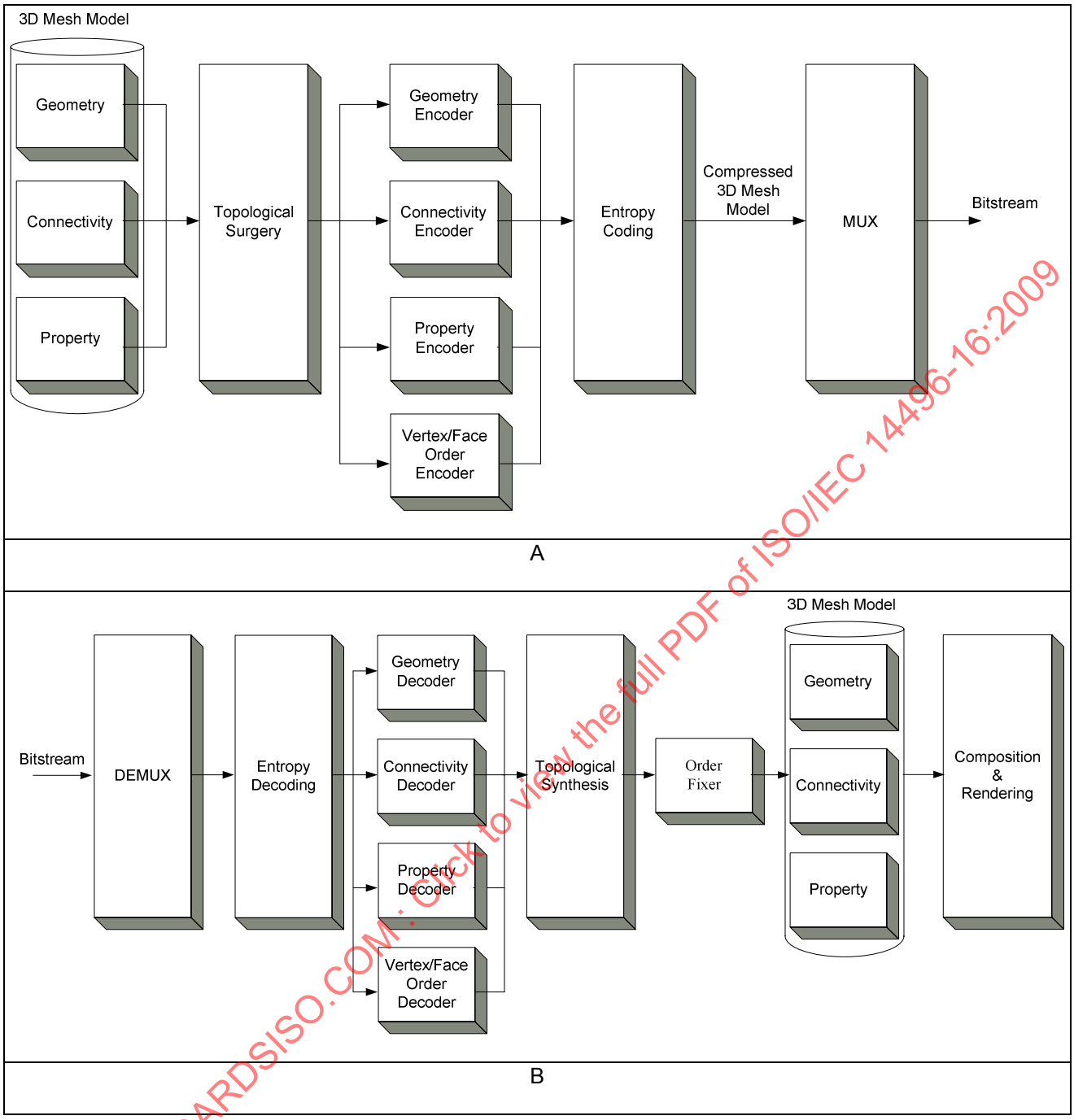


Figure 33 — General block diagram of the 3D mesh compression. A: 3D mesh encoder. B: 3D mesh decoder.

4.2.1.2 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.

4.2.1.3 Bitstream syntax

4.2.1.3.1 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)		
}		

4.2.1.3.2 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()		
}		

4.2.1.3.3 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()		
}		

4.2.1.3.4 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()		
}		

4.2.1.3.5 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
}		
}		

4.2.1.3.6 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
}		
}		
}		

4.2.1.3.7 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
}		
}		
}		

4.2.1.3.8 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
}		
}		
}		

4.2.1.3.9 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()		
else if (function_type == “Multiple_attribute_mode”)		
multiple_attribute_mode_header()		
} while(function_type != “Escape_mode”)		
}		

4.2.1.3.10 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
}		

4.2.1.3.11 Adaptive_quant_texCoord_mode_header

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

4.2.1.3.12 multiple_attribute_mode_header

multiple_attribute_mode_header(){	No. of bits	Mnemonic
number_of_texCoord	5	
for 2 to number_of_texCoord		vlclbf
texCoord_header()		
number_of_otherAttr	8	
for 1 to number_of_otherAttr		
otherAttr_header()		
}		

4.2.1.3.13 otherAttr_header

otherAttr_header() {	No. of bits	Mnemonic
otherAttr_binding	2	uimsbf
otherAttr_dimension	8	
if (otherAttr_binding != “not_bound”) {		
otherAttr_bbox	1	
if (otherAttr_bbox == ‘1’) {		
for (i= &; i < otherAttr_dimension; i++)		
otherAttr_min(i)	32	bslbf
otherAttr_size	32	bslbf
}		
otherAttr_quant	5	uimsbf
otherAttr_pred_type	2	uimsbf
if (otherAttr_pred_type==”tree_prediction” || otherAttr_pred_type==”parallelogram_prediction”) {		
otherAttr_nlambda	2	uimsbf
for (i=1; i<otherAttr_nlambda; i++)		
otherAttr_lambda	4-19	simsbf
}		
}		
}		

4.2.1.3.14 connected_component

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

4.2.1.3.15 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)		
}		

4.2.1.3.16 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--		
}		
}		

4.2.1.3.17 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)		
}		

4.2.1.3.18 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()		
for (i =1; i<= number_of_texCoord; i++)		
root_texCoord(i)		
for (i = 1; i<= number_of_otherAttr; i++)		
root_otherAttr(i)		
}		

4.2.1.3.19 root_coord

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()		
}		
}		

4.2.1.3.20 root_normal

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()		
}		
}		
}		

4.2.1.3.21 root_color

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()		
}		
}		
}		

4.2.1.3.22 root_texCoord

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()		
}		
}		

4.2.1.3.23 root_otherAttr

root_otherAttr() {	No. of bits
if (otherAttr_binding != “not_bound”)	
if (3D_MOBL_start_code == “partition_type_2”) {	
if (otherAttr_binding != “bound_per_vertex” || visited[vertex_index] == 0) {	
root_otherAttr_sample()	
if (otherAttr_binding!= “bound_per_vertex” || visited[vertex_index] == 0) {	
otherAttr_sample()	
otherAttr_sample()	
}	
}	
else {	
root_otherAttr_sample()	
otherAttr_sample()	
otherAttr_sample()	
}	
}	

4.2.1.3.24 root_otherAttr_Sample

'root_otherAttr_sample() {	No. of bits
for (i=0; i<otherAttr_order; i++)	
for (j=0; j<otherAttr_quant; j++)	
qf_decode(**otherAttr_bit**, zero_context)	
}	

4.2.1.3.25 other_Attr_sample

otherAttr_sample() {	No. of bits
for (i=0; i< otherAttr_order; i++) {	
j=0	
do {	
qf_decode(**otherAttr_leading_bit**, otherAttr_leading_bit_context[2*j+i])	
j++	
} while (j<otherAttr_quant && otherAttr_leading_bit == '0')	
if (otherAttr_leading_bit == '1') {	
qf_decode(**otherAttr_sign_bit**, zero_context)	
do {	
qf_decode(**otherAttr_trailing_bit**, zero_context)	
} while (j<otherAttr_quant)	
}	
}	
}	

4.2.1.3.26 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')		vlclbf
qf_decode(**polygon_edge**, polygon_edge_context[**polygon_edge**])		
coord()		
normal()		
color()		
for (i=1; i<= number_of_texCoord; i++)		
texCoord(i)		
for (i=1; i<= number_of_otherAttr; i++)		
otherAttr(i)		
}		

4.2.1.3.27 coord

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()		
}		
}		

4.2.1.3.28 normal

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()		
}		
}		

4.2.1.3.29 color

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()		
}		
}		

4.2.1.3.30 texCoord

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()		
}		
}		

4.2.1.3.31 root_coord_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
}		

4.2.1.3.32 root_normal_sample

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
}		

4.2.1.3.33 root_color_sample

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
}		

4.2.1.3.34 root_texCoord_sample

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
}		

4.2.1.3.35 coord_sample

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)		
}		
}		
}		

4.2.1.3.36 normal_sample

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)		
}		
}		
}		

4.2.1.3.37 color_sample

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)		
}		
}		
}		

4.2.1.3.38 texCoord_sample

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)		
}		
}		
}		

4.2.1.3.39 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
}		
}		
}		
}		

4.2.1.3.40 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()		
}		
}		

4.2.1.3.41 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
}		
}		
}		

4.2.1.3.42 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
}		
}		
}		

4.2.1.3.43 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
}		
}		

4.2.1.3.44 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
}		
}		

4.2.1.3.45 Visual bitstream semantics

4.2.1.3.46 3D Mesh Object

4.2.1.3.47 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'.

4.2.1.3.48 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.

4.2.1.3.49 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.

4.2.1.3.50 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 9 — 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.

4.2.1.3.51 coord_header

coord_binding: This 2 bit unsigned integer indicates the binding of vertex coordinates to the 3D mesh as specified in the following table.

Table 10 — 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 as specified in the following table.

Table 11 — 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 (as specified in the following table).

Table 12 — 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.

4.2.1.3.52 normal_header

normal_binding: This 2 bit unsigned integer indicates the binding of normals to the 3D mesh. The admissible values are described in the following table.

Table 13 — 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. The following table shows the admissible values.

Table 14 — Admissible values for normal_pred_type

normal_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

The following table shows admissible values as a function of normal_binding.

Table 15 — 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. The following table shows admissible values as a function of **normal_pred_type.**

Table 16 — 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.

4.2.1.3.53 color_header

color_binding: This 2 bit unsigned integer indicates the binding of colors to the 3D mesh. The following table shows the admissible values.

Table 17 — 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. The following table shows the admissible values.

Table 18 — Admissible values for color_pred_type

color_pred_type	prediction type
00	no_prediction
01	tree_prediction
10	parallelogram_prediction
11	reserved

The following table shows admissible values as a function of color_binding.

Table 19 — 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. The following shows admissible values as a function of **normal_pred_type.**

Table 20 — 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.

4.2.1.3.54 texCoord_header

texCoord_binding: This 2 bit unsigned integer indicates the binding of texture coordinates to the 3D mesh.The following table describes the admissible values.

Table 21 — 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. The following table shows the admissible values.

Table 22 — Admissible values for texCoord_pred_type

texCoord_pred_type	prediction type
00	no_prediction
01	forbidden
10	parallelogram_prediction
11	reserved

The following table shows admissible values as a function of texCoord_binding.

Table 23 — 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. The following table shows admissible values as a function of **texCoord_pred_type.**

Table 24 — 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.

4.2.1.3.55 3DMC_extension_header

function_type: This 4-bit unsigned integer indicates the function type supported in 3DMC extension. The following table shows the admissible values for **function_type.**

Table 25 — Admissible values for function type

function_type_code	function_type
0000	Order_mode
0001	Adaptive_quant_texCoord_mode
0010	Multiple_attribute_mode
0011~1110	Reserved
1111	Escape code

4.2.1.3.56 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.

4.2.1.3.57 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 multiple_attribute_mode_header

number_of_texCoord: This 5 bit unsigned integer gives the number of texture coordinates per vertex.

number_of_otherAttr: This 8 bit unsigned integer gives the number of additional attributes per vertex

5.9.2.3.1.13 otherAttr_header

otherAttr_binding: This 2 bit unsigned integer indicates the binding of texture coordinates to the 3D mesh. The following table describes the admissible values.

Table 26 — Admissible values for otherAttr_binding

otherAttr_binding	Binding
00	not_bound
01	Bound_per_vertex
10	Forbidden
11	Bound_per_corner

otherAttr_order: This 8-bit unsigned integer gives the order of the other attribute.

otherAttr_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 **otherAttr_min(i)**=0 **otherAttr_size**=1.

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

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

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

otherAttr_pred_type: This 2-bit unsigned integer indicates how colors are predicted. The following tables show the its admissible values, and admissible values as a function of otherAttr_binding, respectively.

Table 27 — Admissible values for otherAttr_pred_type

otherAttr_pred_type	prediction type
00	no_prediction
01	Forbidden
10	parallelogram_prediction
11	Reserved

Table 28 — Admissible combinations of otherAttr_binding and otherAttr_pred_type

otherAttr_binding	otherAttr_pred_type
not_bound	not coded
bound_per_vertex	no_prediction, parallelogram_prediction
bound_per_corner	no_prediction, tree_prediction

otherAttr_nlambda: This 2-bit unsigned integer indicates the number of ancestors used to predict normals. Admissible values of **otherAttr_nlambda** are 1, 2, and 3. The following table shows admissible values as a function of **otherAttr_pred_type.**

Table 29 — Admissible values for otherAttr_nlambda as a function of otherAttr_prediction type

otherAttr_pred_type	otherAttr_nlambda
not_prediction	not coded
tree_prediction	1, 2, 3
parallelogram_prediction	3

otherAttr_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 **otherAttr_quant** + 3. The 3 leading bits represent the integer part, and the **otherAttr_quant** remaining bits the fractional part.

4.2.1.3.58 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.

4.2.1.3.59 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 next table) is arithmetic coded. If the variable openloops is equal to **vg_loop_index**, the trailing '1' in the unary representation is omitted.

Table 30 — 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 next table) is arithmetic coded.

Table 31 — 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.

4.2.1.3.60 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 next table) is arithmetic coded.

Table 32 — 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.

4.2.1.3.61 triangle

td_orientation: This boolean value informs the decoder the traversal order of tt/td pair at a branch. This field is arithmetic coded. The following table shows the admissible values.

Table 33 — 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.

otherAttr _bit: This boolean value indicates the value of a otherAttr bit. This field is arithmetic coded.

otherAttr_leading_bit: This boolean value indicates the value of a leading otherAttr bit. This field is arithmetic coded.

otherAttr_sign_bit: This boolean value indicates the sign of a otherAttr sample. This field is arithmetic coded.

otherAttr_trailing_bit: This boolean value indicates the value of a trailing otherAttr bit. This field is arithmetic coded.

4.2.1.3.62 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.

4.2.1.3.63 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.

4.2.1.3.64 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.

4.2.1.3.65 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.

4.2.1.3.66 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.

4.2.1.4 The decoding process

4.2.1.4.1 3D Mesh Object Decoding

The Topological Surgery decoder is composed of eight main modules, as shown in Figure 34, 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.

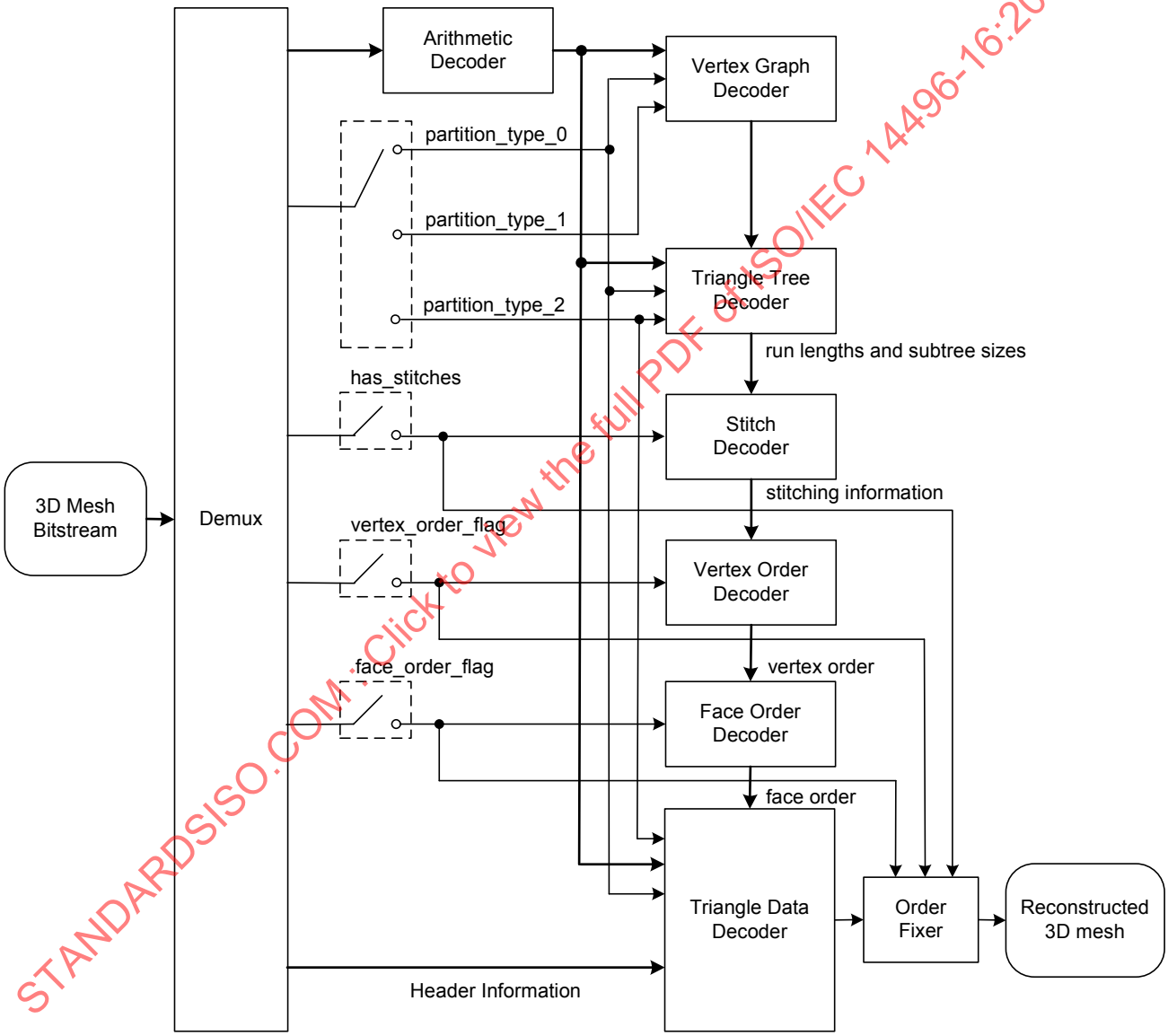


Figure 34 — Block diagram of the Topological Surgery decoder

4.2.1.4.2 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 needs not be applied to the portions of the bit stream that contain arithmetic coded data.

4.2.1.4.3 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.

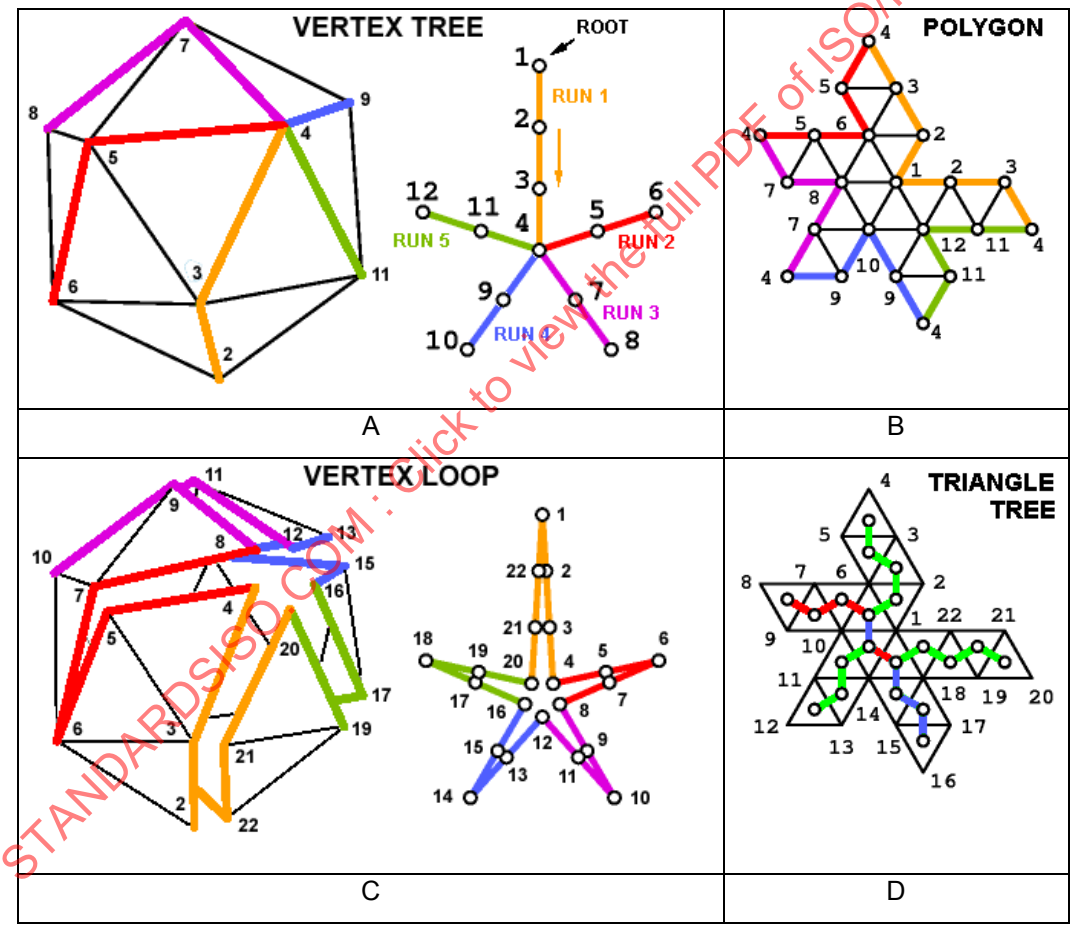


Figure 35 — Topological Surgery representation of a simple mesh.
A: for a simple mesh, the vertex graph is a tree.
B: when a simple mesh is cut through the vertex tree, the result is a simple polygon.
C: each edge of the vertex tree corresponds to exactly two boundary edges of the simple polygon.
D: 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
      }
   }
}
```

4.2.1.4.4 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.

4.2.1.4.5 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.

4.2.1.4.6 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. The following table lists the valid combinations. The binding of each property is obtained from coord_binding, normal_binding, color_binding and texCoord_binding, respectively.

Table 34 — 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.

4.2.1.4.7 Extension Properties

4.2.1.4.8 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.

4.2.1.4.9 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).

4.2.1.4.10 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 36.

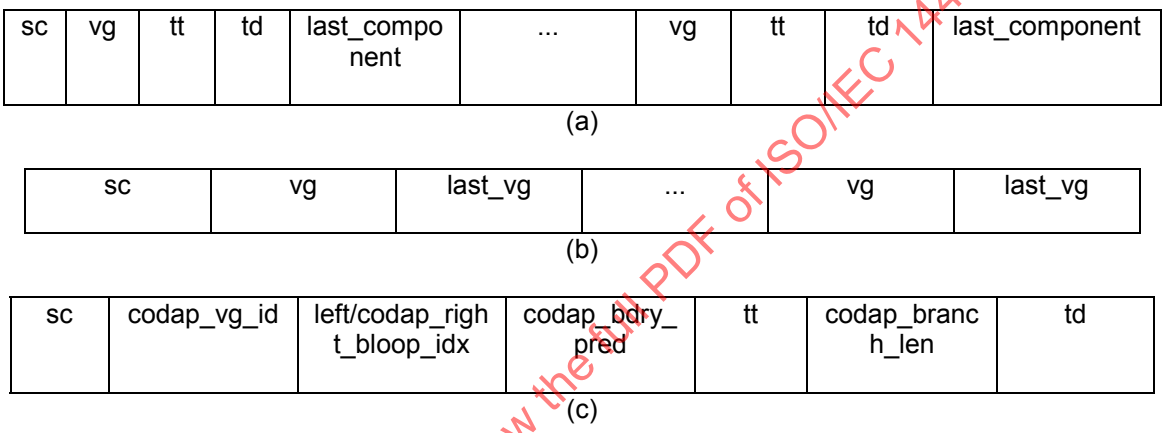


Figure 36 — Types of data partition: (a) one or more group of vg, tt and td included in a partition. (b) one or more vgs included in a partition. (c) the partition has one pair of tt and td.

4.2.1.4.11 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.

4.2.1.4.12 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.

4.2.1.4.13 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.

4.2.1.4.14 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 t1 and t2, are available) then
if (both ancestors are edge distance 1 from current vertex) then
d' = (t1 + t2)/2
else if (t1 is edge distance 1 from current vertex) then  d' = t1
else d' = t2
else  d' = f(a,b,c).
```

4.2.1.4.15 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.

4.2.1.4.16 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.

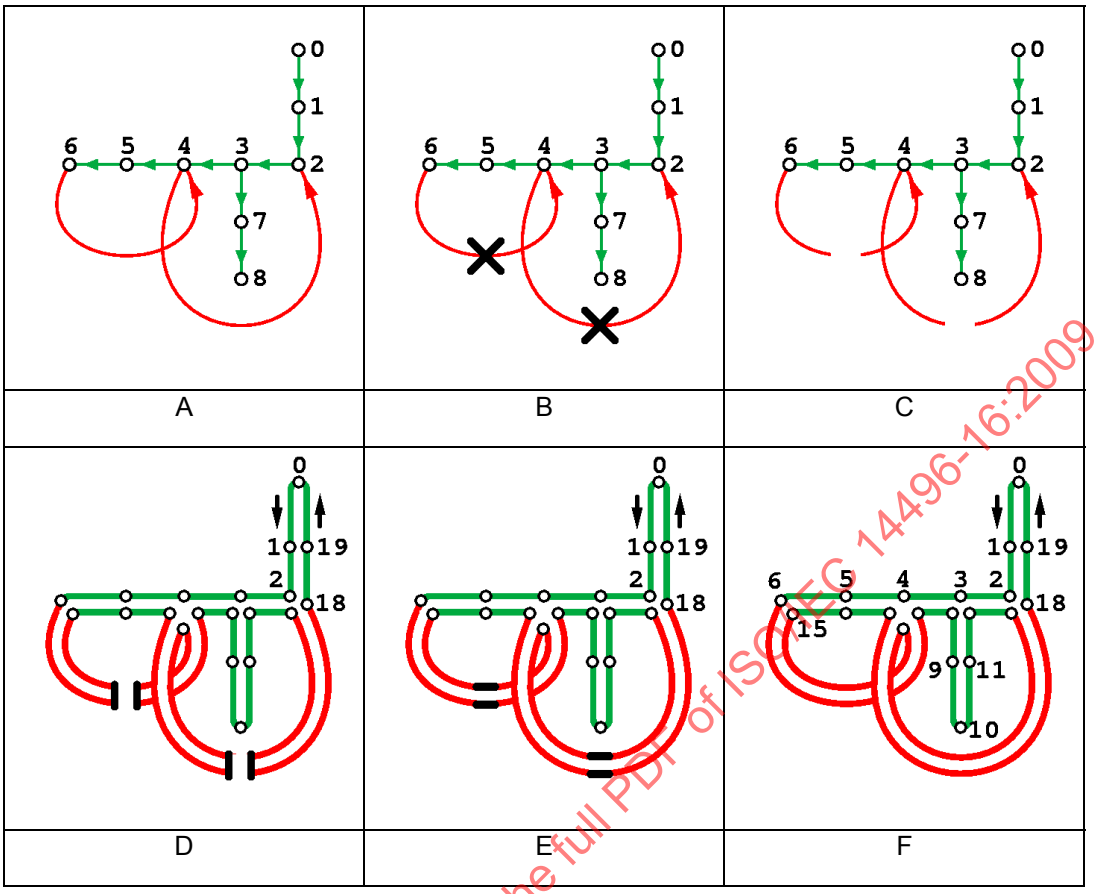


Figure 37 — Steps to build the bounding loop from the vertex graph.
A: vertex graph decomposed into vertex tree (green) and jump edges (red).
B: extended vertex tree is created by cutting jump edges in half.
C: the extended vertex tree has two leaves for each jump edge.
D: build the extended vertex tree 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.

4.2.1.4.17 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.

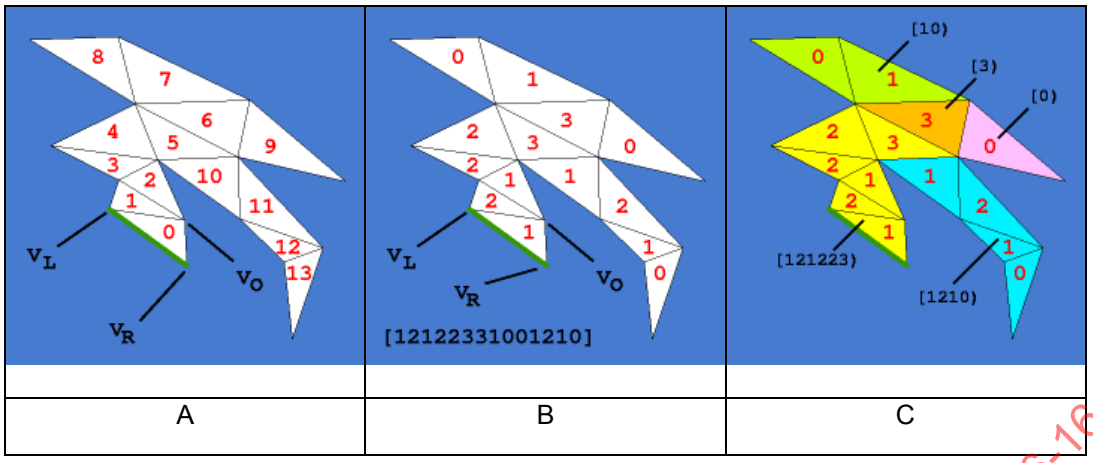


Figure 38 — Representing a simple polygon as a table of triangle runs.
A: depth-first order of traversal of the triangle tree starting from the root edge (green).
B: each triangle is classified by a 2-bit code as leaf(0), advance-right (1), advance-left (2), or branching (3).
C: The sequence of 2-bit 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.

4.2.1.4.18 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 the following figure.

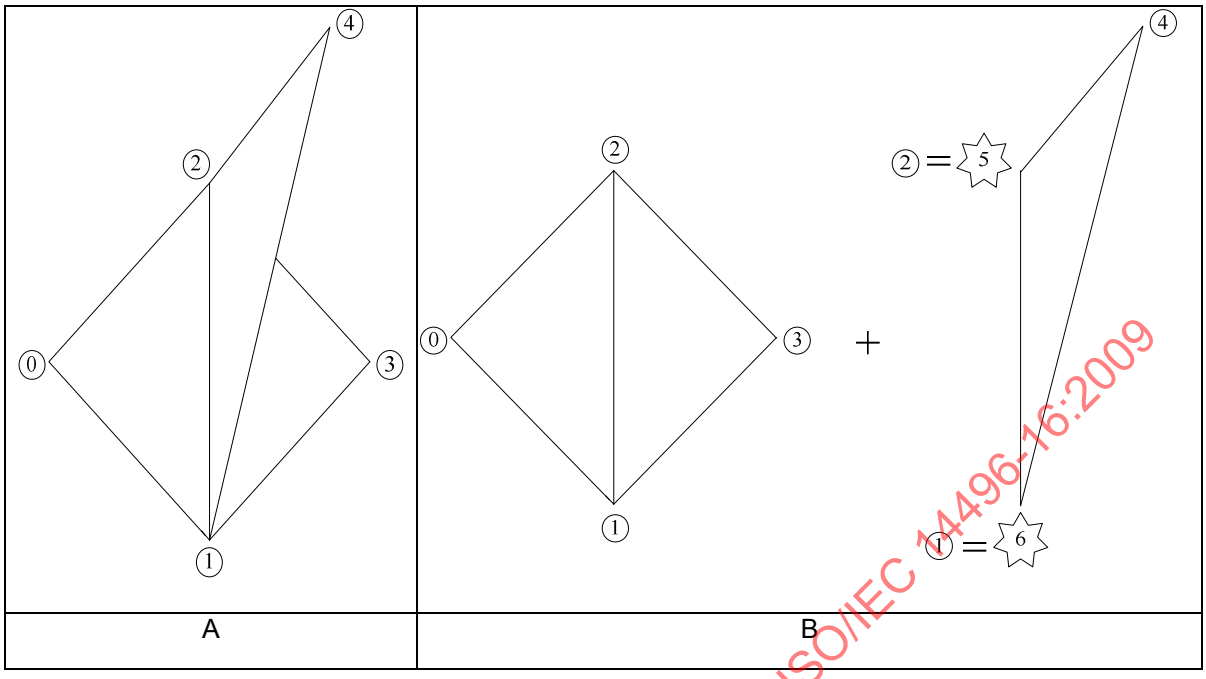


Figure 39 — A non-manifold 3D mesh (A) and converted manifold meshes (B)

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 35 — 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.

4.2.1.4.19 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 40 — vertex order decoder pseudo-code

4.2.1.4.20 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.

To transform the calculated vertex order represented at the unit of connected component into the final vertex order represented at the unit of IndexedFaceSet, following method can be used:

```
void vo_transform_into_IFS(unsigned int vo_id){

// assume the firstVID and vo_offset was given after decoding the
vertex_order_per_CC_header
```

```
for(i=0;i<nV;i++){
   for(j=0;j<nVOffset;j++) {
      if( (vo_id >= firstVID[j]) && vo_id < firstVID[j+1]))
         final_vo_id = vo_id + vo_offset[j];
      }
   }
}
```

4.2.1.4.21 Face Order Decoder

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

```
// face order decoder

int nF = number of faces of the current connected component;

// compute the number of bits(bpfi) required to decode the face order in the
initial state
init_bpfi = bpfi=bitsToRepresent(nF);

// faceVector: for containing not decoded face indices
vector <int> faceVector;

// initialize faceVector to all face indices
for(int q = 0; q < nF; q++)
faceVector.push_back(q);

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

// delList : for containing decoded face indices
int* delList = new int[DecodingFaces];

// compute remaining faces after decoding the current distinguishable unit
nF=nF-DecodingFaces;
for(j=nF;j>0;j--) {
    fo_decode(fo_id, bpfi);
}

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

// update faceVector by deleting decoded fo_id
for(q = 0; q < delCnt; q++)
faceVector.erase(faceVector.begin() + delList[q] - q);

}

// last face order is calculated by the remaining one in faceVector
qFaceIndex.push_back(faceVector[0]);
```

Figure 41 — face order decoder pseudo-code

4.2.1.4.22 Decoding the symbol of the face order

After reading the bpfi bits from the bit stream, the face order is calculated as follows: If bpfi is same as init_bpfi, bpfi is transformed into unsigned integer value and this unsigned integer value becomes the face

order. If bpfi is not same as init_bpfi, bpfi is transformed into unsigned integer value but this unsigned integer value is not the face order. The face order is the (unsigned inter value)-th value among the remaining consecutive face indices in the range 0, …, (F-1).

```
void fo_decode(unsigned int fo_id, bpfi){
ibstrm.getMOBitInt(bpfi, FaceIndex);
if (bpfi==init_bpfi){
   fo_id=FaceIndex
   }
   else {
   // faceVector[FaceIndex] is the (FaceIndex)-th fo_id that does not decoded
      fo_id = faceVector[FaceIndex];
   }
qFaceIndex.push_back(fo_id);
// add fo_id to delList for updating faceVector
delList[delCnt++] = fo_id;
}
```

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

To transform the calculated face order represented at the unit of connected component into the final face order presented at the unit of IndexedFaceSet, following method can be used:

```
void fo_transform_into_IFS(unsigned int fo_id){

// assume the firstFID and fo_offset was given after decoding the face_order_per_CC_header

for(i=0;i<nF;i++)
{
   for(j=0;j<nFOffset;j++)
   {
      if( (fo_id >= firstFID[j]) && fo_id < firstFID[j+1]))
         final_fo_id = fo_id + fo_offset[j];
   }
}
}
```

4.2.1.4.23 Triangle Data Decoder

The triangle data decoder operates in three steps. It first decodes a header, then a root triangle, which is followed by a sequence of triangles. Note that this sequence is empty if there is a single triangle in the current connected component.

The header is a boolean flag **triangulated** that indicates if all faces in the connected component are triangles. If this flag indicates true, the **polygon_edge** field is not coded.

The decoding of a root triangle and of a non-root triangle are the same with the following exception. The **polygon_edge** field is never coded for a root triangle of a connected component, and the number of property samples may differ.

Two fields indicate connectivity information for the current triangle: **marching_edge** and **polygon_edge**. **marching_edge** is not coded for branching and leaf triangles. **polygon_edge** is not coded for the root triangle or if the **triangulated** flag is set true. However in a partition of partition_type_2, the **polygon_edge** is coded for the root triangle if the **triangulated** field is set false.

The number of per vertex property samples may differ depending on the boundary prediction mode and visited vertices. If the boundary prediction mode is restricted, then the two starting points are already visited in the previous partition and only one root sample is coded. However, if the remaining vertex is also visited, there are no samples coded. If the boundary prediction mode is extended, the root triangle always has all 3 samples.

Table 36 — Number of samples to decode in a root triangle for each binding

Binding	# samples	Detail
None	0	
per vertex	0, 1, or 3	1 root and 2 non-root samples
per face	1	1 root sample
per corner	3	2 root and 1 non-root samples

For non-root triangles, the triangle data decoder decodes zero or one sample for each property bound per vertex. If the opposite vertex of the current triangle has been visited before, the value of the sample is already known and does not need to be decoded again. If the opposite vertex of the current triangle has never been visited before, then a sample is read. The bounding loop look-up table and the sizes of subtrees of the triangle tree are necessary to determine whether the opposite vertex of the current triangle has been visited previously. The boundary prediction mode is necessary to determine whether to consider the visited vertices within the partition or including every vertices visited in the previous partitions.

Table 37 — Number of samples to decode in a non-root triangle for each binding. The number of samples may be conditioned.

Binding	# samples	Condition
None	0	
per vertex	0 or 1	vertex never visited
per face	0 or 1	polygon_edge is set
per corner	1 or 3	polygon_edge is set

The traversal order is fixed in basic topological surgery. However, in order to change traversal order, **td_orientation** field is coded at a branching triangle on the main stem of the triangle tree in the partition. In other words, if the number of previously decoded branching triangles is equal to that of leaf triangles within the current partition, i.e. the depth value is zero, the **td_orientation** field is coded.

4.2.1.4.24 Decoding a root sample

A root sample is the first decoded sample in a root triangle. If there are several samples in a root triangle only the first sample is a root sample. The remaining samples in the root triangle are non-root samples.

The **coord_quant** bits of each coordinate of a root geometry sample are received starting with the most significant.

4.2.1.4.25 Decoding a non-root sample

A coordinate of a non-root geometry sample is composed of **coord_quant** bits, plus a sign bit. The coordinate is coded bit by bit starting from the most significant. The leading 0's, if any, and the leftmost 1, if any, are **coord_leading_bit**'s are coded using an adaptive probability estimation model. If not all **coord_leading_bits** are 0's a **coord_sign_bit** is coded. Finally **coord_quant** minus the number of **coord_leading_bits** **coord_trailing_bits** are coded using a fixed probability model.

3***coord_quant** probability estimation models are defined to code the **coord_leading_bit** field. The probability model is selected according to the bit position and to the coordinate number.

The decoded value is not the actual value of the sample, but a correction that has to be applied to the prediction to obtain the reconstructed sample. The reconstructed sample is equal to the sum of the prediction and of the decoded value. The computation of the prediction is described below.

It may happen that a reconstructed value is outside the bounding box, in which case a masking operation is carried out to insure that the final reconstructed value is indeed inside the bounding box. The masking operation depends on the quantisation step **coord_quant**. The **coord_quant** less significant bits of the reconstructed value are kept as is, and all other (more significant) bits are cleared.

The same structure applies to the other properties.

4.2.1.4.26 Property Prediction

For increased coding efficiency, the value of a property is coded relative to a prediction. This subclause describes the computation of the prediction for each type of binding. The following two tables describe all the valid combinations of property binding, property type, and prediction type.

Table 38 — Valid property bindings for each property type

binding	meaning	coord	normal	color	texCoord
00	not encoded		X	X	X
01	per vertex	X	X	X	X
10	per face		X	X	
11	per corner		X	X	X

Table 39 — Valid combinations of prediction type and property binding

pred_type	nlambda	binding
no_prediction	not coded	bound_per_vertex, bound_per_face, or bound_per_corner
tree_prediction	1	bound_per_face or bound_per_corner
parallelogram_prediction	3	bound_per_vertex

For tree and parallelogram prediction, the prediction of the properties in the current triangle are always based on the values in the previous triangle in the triangle tree. The prediction is weighted by the lambda coefficients decoded by the header decoder.

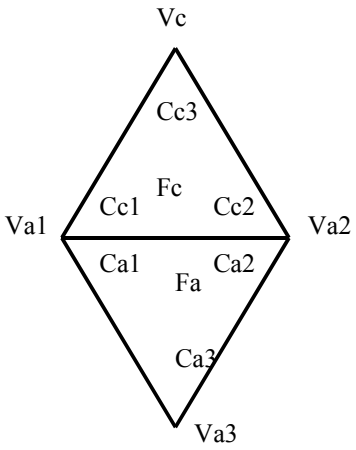


Figure 42 — Prediction of property values.
The value at the current vertex Vc is predicted from the values at ancestor vertices Va1, Va2 and Va3.
The value at the current face Fc is predicted from the value at ancestor face Fa.
The value at the current corner Cc1 is predicted from the value at ancestor corner Ca1.
The value at current corner Cc2 is predicted from the value at ancestor corner Ca2.
The value at current corner Cc3 is predicted from the value at current corner Cc1.

4.2.1.4.27 Tree prediction

If the property is bound per face, the prediction of Fc is equal to Fa. If the current triangle is the root of the triangle tree, Fc is not predicted.

If the property is bound per corner, the prediction of Cc1 is equal to Ca1, the prediction of Cc2 to Ca2, and the prediction of Cc3 to Cc1. If the current triangle is the root of the triangle tree, Cc1 is not predicted, Cc2 is predicted by Cc1, and Cc3 by Cc1.

4.2.1.4.28 Parallelogram prediction

If the property is bound per vertex, the prediction of Vc is equal to lambda1*Va1 + lambda2*Va2 + lambda3*Va3. If the current triangle is the root of the triangle tree, Va1 and Va2 must be encoded too, Va1 is not predicted, Va2 is predicted by Va1, and Vc by Va1.

4.2.1.4.29 Inverse quantisation

There are three inverse quantisation procedures. The first one applies to geometry, colors and texture coordinates. The second applies to normals and the last one applies to texture coordinate.

Given a quantised geometry sample (qx, qy, qz), the reconstructed geometry sample (x,y,z) is obtained as:

```
x = coord_xmin + coord_size*xq/((1<<coord_quant)-1)
y = coord_ymin + coord_size*yq/((1<<coord_quant)-1)
z = coord_zmin + coord_size*zq/((1<<coord_quant)-1)
```

The procedure for colors and texture coordinates is the same.

There is a special inverse quantisation procedure for normals. Normals are always quantised to an odd number of bits **normal_quant**. The 3 most significant bits of **normal_quant** indicate the octant the normal lies in, and the remaining bits define an index within the octant.

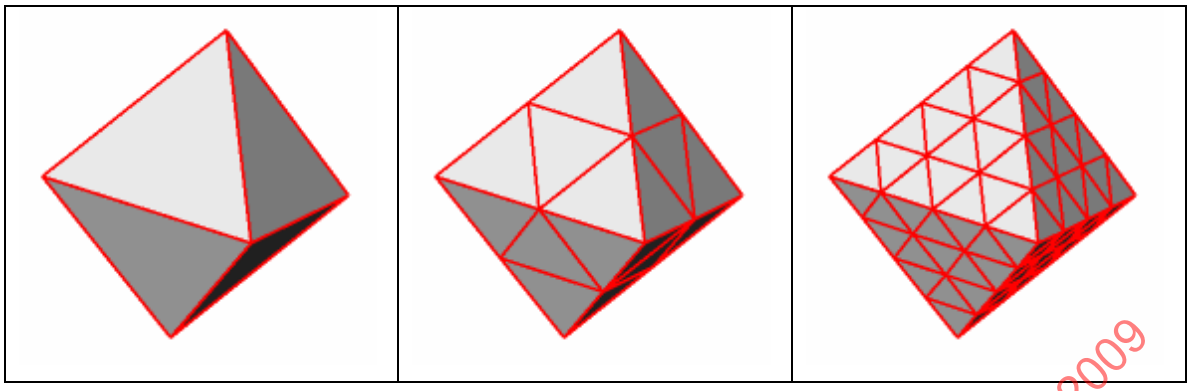


Figure 43 — The quantisation of normals is based on the hierarchical subdivision of the unit octahedron. Left: unit octahedron. Middle: unit octahedron after one subdivision. Right: unit octahedron after two subdivisions.

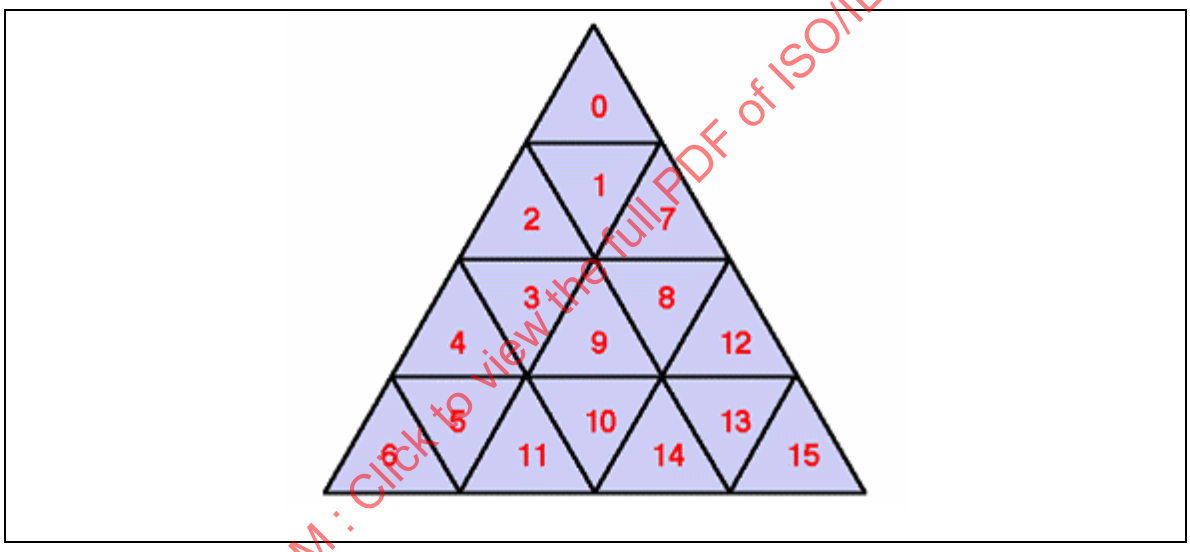


Figure 44 — Numbering of the triangles within an octant. Note that the top corner corresponds to |z|=1, the leftmost corner to |x|=1 and the rightmost corner to |y|=1.

Given an index i, and the 3 sign bits sx, sy and sz, the normal (x,y,z) can be analytically reconstructed as follows:

```
nbins = 1 << ((normal_quant - 3) / 2);

y0 = nbins - ceil(sqrt(nbins*nbins - i));
x0 = i + y0*y0;
skew = (x0 & 1)*2.0/3.0;
x0 = (x0 >> 1) & (nbins - 1);

x = (float)x0 + skew;
y = (float)y0 + skew;
z = (float)nbins - x - y;

n = 1.0/sqrt(x*x + y*y + z*z);
x = (sx) ? -x*n : x*n;
y = (sy) ? -y*n : y*n;
z = (sz) ? -z*n : z*n;
```

If the efficient texture mapping of 3DMC extension is supported, there is the third inverse quantisation procedure for texture coordinates, which uses two values, **texCoord_quant_u** and **texCoord_quant_v**. Given a quantised texture coordinates sample (qu, qv), the reconstructed texture coordinates sample (u,v) is obtained as:

```
u = qu/texCoord_quant_u
v = qv/texCoord_quant_v
```

4.2.1.4.30 Arithmetic decoder

This subclause describes the QF-coder for arithmetic decoding. C++-style routines are provided. The arithmetic decoder relies on a set of variables that are described in the table below.

Register	Description
A	size of current interval
C	current arithmetic code
count	number of renormalisations before reading next byte
symbol	value of decoded symbol
Qe	probability estimate for least probable symbol
MPS	value of most probable symbol

The registers Qe and MPS are dependent upon a context. For each context, there is a state variable including the Qe and MPS registers.

4.2.1.4.31 Initialisation

The arithmetic coder reads data from the input on a per byte basis. Therefore the stream pointer shall be byte aligned before initialising the arithmetic decoder.

In the initialisation process, all variables are initialised to proper values. 2 bytes are read from the input stream into register C.

```
void qf_start() {
  A = 0x00010000;
  C = 0x00010000;
  count = 0;
  sw_renorm_decode();
}
```

4.2.1.4.32 Decoding a symbol

When a binary symbol is decoded, the interval is split into two. The half in which C lies determines the value of the bit. The interval is then reduced to the corresponding half. If the size of the new interval is smaller than 0x80000000, then the interval is renormalised and the probability estimator updated.

```
void qf_decode(int *symbol, QState *state) {
  *symbol = state->MPS;
  A -= state->Qe;
  if (C <= A) {
    if (A & 0x80000000)
      return;
```

```
    *symbol ^= A < state->Qe;
  }
  else {
    *symbol ^= A >= state->Qe;
    C = C - A;
    A = state->Qe;
  }
  state->update(*symbol);
  renorm_decode();
}
```

4.2.1.4.33 Renormalisation

In the renormalisation step, the size of the interval is doubled until it reaches at least 0x80000000. Each time the size of the interval is doubled a bit is consumed. When no more bit is available for consumption in the less significant bits of the register C, a new byte is retrieved from the stream.

```
void qf_renorm() {
  do {
    if (count == 0) {
      int b = get_byte();
      C -= 0xff << 8;
      C += b << 8;
   count = (b == 0) ? 7 : 8;
    }
    count--;
    C = C << 1;
    A = A << 1;
  } while (!(A & 0x80000000));
}
```

4.2.1.4.34 Probability estimation

A majority of the fields that are arithmetic coded benefit from adaptive probability estimation. This adaptive estimation is defined by a Markov Model. Each state of the model defines a probability of the LPS, a next state if the currently coded symbol is the MPS, a next state if the currently coded symbol is the LPS, and a flag that indicates if the value of the MPS should be changed if the currently coded symbol is the LPS. The Markov model to be used is defined in Table 40. It is named FA-JPEG for Fast-Attack JPEG, and has been designed such as to minimize the number of states.

A fixed probability of the LPS and a fixed MPS are defined for the fields for which there is no adaptive probability estimation.

The next table lists the probability estimation used for each of the fields.

The models are global and initialised to index = 0, MPS = 0 before the first connected component.

The context of order 1 models is reset before each component.

The state is updated every time the renormalisation procedure is called.

Table 40 — List of fields that are arithmetic coded and their associated probability estimation

Field	probability estimation	context
last_component	FA-JPEG	last_component_context
codap_last_vg	FA-JPEG	codap_last_vg_context
codap_vg_id	FA-JPEG	zero_context

codap_left_bloop_idx	FA-JPEG	zero_context
codap_right_bloop_idx	FA-JPEG	zero_context
codap_bdry_pred	FA-JPEG	zero_context
vg_simple	FA-JPEG	vg_simple_context
vg_last	FA-JPEG	vg_last_context
vg_forward_run	FA-JPEG	vg_forward_run_context
vg_loop_index	FA-JPEG	vg_loop_index_context
vg_run_length	FA-JPEG	vg_run_length_context
vg_leaf	FA-JPEG	vg_leaf_context
vg_loop	FA-JPEG	vg_loop_context
tt_run_length	FA-JPEG	tt_run_length_context
tt_leaf	FA-JPEG	tt_leaf_context
codap_branch_len	FA-JPEG	codap_branch_len_context
triangulated	FA-JPEG	triangulated_context
marching_edge	FA-JPEG, order 1, initial context = 1	marching_edge_context[0..1]
td_orientation	FA-JPEG	td_orientation_context
polygon_edge	FA-JPEG, order 1, initial context = 1	polygon_edge_context[0..1]
coord_bit	fixed Qe = 0x5601, MPS = 0	zero_context
coord_leading_bit	FA-JPEG, 3*coord_quant contexts	coord_leading_bit_context[0..3*coord_quant-1]
coord_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
coord_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
normal_bit	fixed Qe = 0x5601, MPS = 0	zero_context
normal_leading_bit	FA-JPEG, normal_quant contexts	normal_leading_bit_context[0..normal_quant-1]
normal_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
normal_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_leading_bit	FA-JPEG, 3*color_quant contexts	color_leading_bit_context[0..3*color_quant-1]

color_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
color_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_leading_bit	FA-JPEG, 2*texCoord_quant contexts	textCoord_leading_bit_context[0..2*texCoord_quant-1]
texCoord_sign_bit	fixed Qe = 0x5601, MPS = 0	zero_context
texCoord_trailing_bit	fixed Qe = 0x5601, MPS = 0	zero_context

Table 41 — The Fast Attack JPEG (FA-JPEG) Markov model to estimate probabilities. For each state, we define the probability of the LPS, the next state in the event of an MPS, the next state in the event of an LPS, and a switch MPS flag in the event of an LPS

State	Probability of LPS	Next state if MPS	Next state if LPS	Switch MPS if LPS
0	0x5601	1	1	yes
1	0x3401	2	6	no
2	0x1801	3	9	no
3	0x0ac1	4	12	no
4	0x0521	5	29	no
5	0x0221	38	33	no
6	0x5601	7	6	yes
7	0x5401	8	14	no
8	0x4801	9	14	no
9	0x3801	10	14	no
10	0x3001	11	17	no
11	0x2401	12	18	no
12	0x1c01	13	20	no
13	0x1601	29	21	no
14	0x5601	15	14	yes
15	0x5401	16	14	no
16	0x5101	17	15	no
17	0x4801	18	16	no
18	0x3801	19	17	no
19	0x3401	20	18	no

20	0x3001	21	19	no
21	0x2801	22	19	no
22	0x2401	23	20	no
23	0x2201	24	21	no
24	0x1c01	25	22	no
25	0x1801	26	23	no
26	0x1601	27	24	no
27	0x1401	28	25	no
28	0x1201	29	26	no
29	0x1101	30	27	no
30	0x0ac1	31	28	no
31	0x09c1	32	29	no
32	0x08a1	33	30	no
33	0x0521	34	31	no
34	0x0441	35	32	no
35	0x02a1	36	33	no
36	0x0221	37	34	no
37	0x0141	38	35	no
38	0x0111	39	36	no
39	0x0085	40	37	no
40	0x0049	41	38	no
41	0x0025	42	39	no
42	0x0015	43	40	no
43	0x0009	44	41	no
44	0x0005	45	42	no
45	0x0001	45	43	no

4.2.2 Wavelet Subdivision Surfaces

4.2.2.1 Downstream syntax

This is the syntax of the WaveletSubdivisionSurface downstream.

4.2.2.1.1 WMDecoderConfig

4.2.2.1.1.1 Syntax

```
class WMDecoderConfig extends AFXExtDescriptor : bit(8) tag = 2 {
   bit(1) hasScaleCoeff;
   if (hasScaleCoeff)
      int(5) NbBpSC;
   int(5) NbBPX;
   int(5) NbBPY;
   int(5) NbBPZ;
   int(2) Wtype;
   bit(1) lift;
   int(4) NbLevels;
   float(32) Xmax;
   float(32) Ymax;
   float(32) Zmax;
}
```

4.2.2.1.1.2 Semantics

hasScaleCoeff: this is a flag with value 1 if and only if the scale coefficients are transmitted (otherwise they are supposed to be added to the base mesh vertices).

NbBpSC: this is the number of bits on which the scale coefficients will be read.

NbBPX, **NbBPY**, **NbBPZ**: these are respectively the numbers of bitplanes on which are encoded the magnitude of the first second and third component of the wavelet coefficients, an extra bit being necessary to encode their sign.

Wtype: this is an integer representing the wavelet type. 0 corresponds to midpoint subdivision, 1 to Loop's scheme, 2 to Dyn's butterfly scheme, and 3 is reserved for future use.

lift: this is a flag with value 1 if and only if the high pass filters are lifted [78].

NbLevels: this is the number of subdivision levels of the mesh.

Xmax, **Ymax**, **Zmax**: these are 32 bits floating point values representing the symmetric quantization intervals for the first, second and third coordinate respectively. These values are chosen so that, e.g., the first component of all wavelet coefficients belong to [-Xmax, +Xmax], and are therefore mapped to $[-2^{NbBPX}+1, 2^{NbBPX}-1]$.

4.2.2.1.2 Wavelet_Mesh_Object

4.2.2.1.2.1 Syntax

```
class Wavelet_Mesh_Object {
   bit(1) isInBand;
   if (isInBand)
      WMDecoderConfig WMDecoderConfig;
   bit(1) WMOL;
   bit(1) isInLocalCoordinates;
   if (WMOL && WMDecoderConfig.hasScaleCoeff)
      Wavelet_Mesh_Object_Scale_Coeff Coefficients;
   ReadZT ZeroTree;
}
```

4.2.2.1.2.2 Functionality and semantics

WMOL: a boolean with value 0 if and only if the current stream is a base layer.

isInLocalCoordinates: this is a flag with value 1 if and only if the decoder must reconstruct the mesh considering the wavelet coefficients in local frames.

ZeroTree: this is the SPIHT encoded representation of one or more bitplanes.

The decoding of the wavelet coefficients can start even if no base layer is received. It is then considered that the scale coefficients are all zero and the hierarchy is the default one.

4.2.2.1.3 Wavelet_Mesh_Object_Scale_Coeff

This is the class containing the scale coefficients.

4.2.2.1.3.1 Syntax

```
class Wavelet_Mesh_Object_Scale_Coeff {
   int i;
   for (i=0 ; i< Nb_sc ; i++) {
      int(WMDecoderConfig.NbBpSC) Sc[i];
   }
}
```

4.2.2.1.3.2 Semantics

Nb_sc: is the number of scaling coefficients, known as the number of vertices in the base mesh.

Sc: is an array containing the scale coefficients related to the vertices of the base mesh in the same order than in its description.

4.2.2.1.4 ReadZT

4.2.2.1.4.1 Syntax

```
class ReadZT {
   int FT, LT, i, j, k;
   int(5) LengthNbBits;
   int(5) FBP;
   int(5) LBP;
   bit(1) X;
   bit(1) Y;
   bit(1) Z;
   bit(1) isPartial;
   if (isPartial) {   // Partial transmission
      int(16) ZtId;
      FT = ZtId;
      LT = FT + 1;
   } else {           // Full transmission
      FT = 0;
      LT = NumTree;
   }
   for (j=FBP; j<=LBP ; j++) {
      for (i=FT; i<LT; i++) {
         if (X) {
            int(LengthNbBits) BPLength;
            for (k=0 ; k<BPLength ; k++) {
               bit(1) Bx[i][j][k];
            }
         }
         if (Y) {
            int(LengthNbBits) BPLength;
            for (k=0 ; k<BPLength ; k++) {
               bit(1) By[i][j][k];
            }
```

```
            }
            if (Z) {
               int(LengthNbBits) BPLength;
               for (k=0 ; k<BPLength ; k++) {
                  bit(1) Bz[i][j][k];
               }
            }
         }
      }
   }
}
```

4.2.2.1.4.2 Semantics

NumTree: this is the number of trees, known as the number of edges in the base mesh.

LengthNbBits: this is the number of bits on which **BPLength** is read.

isPartial: this is a flag with value 0 if and only if the totality of the zerotrees are transmitted.

X, **Y**, **Z**: these are three flags with value 1 if bitplanes relative to the first, second and third components follow.

ZtId: this is the number of the currently sent or refined zero-tree.

BPLength: this is the length of the current bitplane.

component: this is a two-bits variable indicating which table (Bx, By or Bz) is updated.

FBP: this is the number of the first bitplane to read. Its value must be greater than or equal to 0 (most significant bitplane) and strictly lower than **NbBPX**, **NbBPY** or **NbBPZ**, depending on the component being read.

LBP: this is the number of the last bitplane to read. Its value must be greater than or equal to **FBP** and strictly lower than **NbBPX**, **NbBPY** or **NbBPZ**, depending on the component being read.

Bx, **By**, **Bz**: these are three 3-tables of bits used for encoding the first, second and third components of the wavelet coefficients. The first entry is the number of the zero-tree. The second one is the number of the current bitplane received. The third one is the number of the received bit in this plane.

NOTE – Decoding separately each component as independent zero-trees enables reception of normal meshes [37]. It also allows reception in different channels or decoding in separate threads.

The decoding process is exposed in Annex A.

4.2.2.2 Upstream syntax (for backchannel)

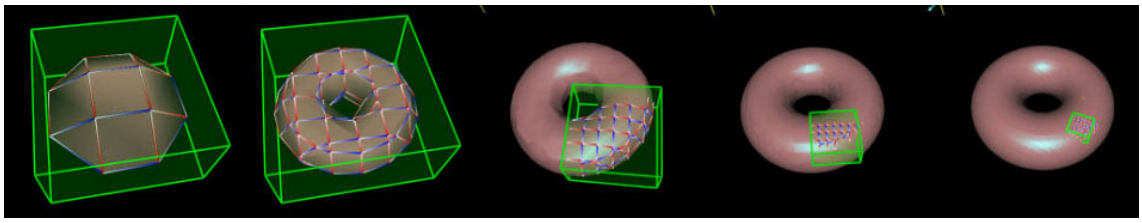
When specified as an upstream in corresponding ES descriptor, the WaveletSubdivisionSurface stream has to be read according to the AFX Generic Backchannel syntax (see Subclause 4.5.2).

4.2.3 MeshGrid stream

4.2.3.1 Overview

The MeshGrid stream has a modular structure consisting of a sequence of parts. There are several types of parts with different semantics, each one identified by a unique tag. All part types are optional, and several parts of the same type can be present in the stream, but their order is not imposed. The following parts of the MeshGrid stream are encoded at each resolution level, either as one single *region of interest* (ROI) or in separate ROIs, if view-dependent decoding is needed: (1) a *connectivity-wireframe description*, (2) a *reference-grid description*, and (3) a *vertices-refinement description* (i.e. refining the position of the vertices relative to the reference-grid – the offsets).

Due to its regular nature, it is straightforward to divide the reference-grid into ROIs, and to encode the surface locally in each of these ROIs. As illustrated in Figure 45, single ROI mode can be mixed with multiple ROIs mode (view-dependent mode) for different resolution levels.



(a) (b) (c) (d) (e)

Figure 45 — Different resolution levels of a multi-resolution MeshGrid can be coded as one mesh (view-independent coding) or split into several ROI (view-dependent coding).

4.2.3.2 The Bitstream structure

The description of the stream is parsed in the *MeshGridDecoderConfig* class defined in subclause 4.2.3.2.2. The MeshGrid stream is parsed in the *MeshGridStream* class. The stream is organized as a succession of resolution level descriptions, starting from level 0 to (*totalNumLevelsMesh*-1), where *totalNumLevelsMesh* represents the total number of resolution levels of the mesh, and is computed in the MeshGridDecoderConfig class.

The mesh description part consists of a header, the *MeshDescriptor*, and a body consisting of three parts: the mesh connectivity coding (parsed in the *MeshConnectivity* class), the vertices' reposition bit plane coding (parsed in the *VertexRefine* class) and the vertices' refinement coding (parsed in the *VertexRefine* class). The grid description part consists of a header, parsed in the *GridDescriptor* class, and a body, parsed in the *GridCoefficients* class. In case the number of resolution levels of the grid (*totalNumLevelsGrid* defined in Annex B.1.3) is lower than the number of resolution levels of the mesh (*totalNumLevelsMesh*), the first *N* resolution level descriptions of the stream will only contain the mesh description part, with *N* defined as the difference between *totalNumLevelsMesh* and *totalNumLevelsGrid*. In case *totalNumLevelsGrid* is larger than *totalNumLevelsMesh*, the resolution '0' description of the stream contains one mesh description and $|N|$ grid descriptions.

The constituent parts in the MeshGrid stream must be decoded in the following order: for any resolution level of the mesh, the *MeshDescriptor* has to be parsed before retrieving any ROI for that resolution level. Decoding the ROIs implies decoding the mesh connectivity, which may be followed by parsing the vertices' reposition or refinement. Similarly, the grid decoding consists of parsing the header of the grid followed by the decoding the grid tiles.

4.2.3.2.1 Global Constants

const unsigned int LEVEL_BITS = 6;	Number of bits allocated for storing the number of resolution levels
const unsigned int REFINE_BITS = 6;	Number of vertices' refine bits
const unsigned int FILTER_BITS = 4;	For specifying the type of filter used for coding the grid.
const unsigned int ROI_BITS = 6;	Number of bits defined for a field specifying the length in bits allocated for the number of ROIs fields for each {u,v,w} direction (nROIs)
const unsigned int MIN_ROI_SIZE = 6;	The minimum ROI size is 5

const unsigned int FIELD_BITS = 6;	Number of bits for coding the log2 of the threshold. Used as well for storing the number of bits allocated for the scale values (gridScale) for each of the {x,y,z} wavelet coefficients.
const unsigned int MIN_SCALE = 1;	The minimum scale value is 1 (gridScale)
const unsigned int QUANT_BITS = 6;	The number of bits for storing the number of quantization bits of the {x,y,z} coordinates of the grid corners
const unsigned int GRID_BITS = 5;	Number of bits defined for a field specifying the length in bits allocated for the counters holding the size of the encoded wavelet coefficients for each {u,v,w} direction
const unsigned int VERTS_BITS = 5;	Number of bits defined for a field specifying the length in bits allocated for the counters holding the size of the encoded connectivity-wireframe and refine bits
const unsigned int MIN_SPLIT_SIZE = 9;	The minimum cube size that can be further split
const unsigned int MIN_DEC_SIZE = 3;	The minimum cube size that consists of several sub-bands
const unsigned int GENERIC_MESH = 0	The encoded mesh is generic, meaning that the triangulation may contain triangles, quadrilaterals, pentagons, hexagons and heptagons
const unsigned int TRI_MESH = 1	The encoded mesh contains only triangles
const unsigned int QUADRI_MESH = 2	The encoded mesh contains only quadrilaterals
const unsigned int HEXA_MESH = 3	The encoded mesh contains only hexagons
const unsigned int READ_REPOSITION_BITS = 2	No default value specified for the reposition bits

4.2.3.2.2 MeshGridDecoderConfig

4.2.3.2.2.1 Syntax

```
class MeshGridDecoderConfig extends AFXExtDescriptor: bit(8) tag=1{
   // the max number of resolution levels
   unsigned int totalNumLevelsMesh;

// number of resolution levels specified for each u,v,w direction
// of the reference-grid.
  FixedUVW nLevelsMesh(LEVEL_BITS);
  totalNumLevelsMesh = max(nLevelsMesh.u, max(nLevelsMesh.v, nLevelsMesh.w));

// number of slices (reference surfaces) in the u,v,w directions
// corresponding to the last resolution level
  ParsableUVW nSlices;
```

```
// number of ROIs (regions of interest) in the u,v,w directions
// corresponding to the last resolution level
  ParsableUVW nROIs;

  // flags
  bit (1) hasConnectivityInfo;
  bit (1) hasRefineInfo;
  bit (1) hasRepositionInfo;
  bit (1) hasGridInfo;

   // reserved
   bit(8) attributes = 0;

   // multilayer
   bit (1) hasMultiLayer;
   if (hasMultiLayer) {
      ParseValue nLayersunsigned int (LAYER_BITS) nBitsLayer;
      unsigned int nBitsLayer = ceil(log2(nLayers)) + 1;
   }

   // type of homogeneous mesh
   unsigned int(2) meshType;

   if (meshType == QUADRI_MESH) {
      if (hasConnectivityInfo) {
      // connectivity bits and uniform quads splitting flags
   bit (1) sameBorderOrientation;
      bit (1) uniformSplit;
     }
     else {
       unsigned int sameBorderOrientation = 1;
       unsigned int uniformSplit = 1;
     }

     if (nSlices.w == 1)
      ParseValue offsetAmplitude;
  }

// the choices for cyclic and folded mesh
bit (3) cyclic_folded;

  // number of refine bits vertex
  RefineVertexDescriptor refine;

  // filter type for grid coding
  unsigned int(FILTER_BITS) filterType;

  // scale values for the x,y,z encoded grid coordinates
  // minimum scale factor is MIN_SCALE
  ScaleXYZ gridScale(FIELD_BITS);

  // the grid corners
  GridCorners gridCorner(nSlices, cyclic_folded);
}
```

4.2.3.2.2.2 Semantics

The **MeshGridDecoderConfig** class initializes the MeshGrid decoder. It **(1)** parses the resolution levels of the mesh (*nLevelsMesh*) for the $\{U,V,W\}$ directions with acceptable values for *nLevelsMesh.u, nLevelsMesh.v* and *nLevelsMesh.w,* lying in the range [1, 63], **(2)** computes the maximum number of resolution levels *totalNumLevelsMesh*, parses **(3)** the number of slices (*nSlices*) $\{S_U,S_V,S_W\}$ and **(4)** the number of ROIs (*nROIs*) corresponding to the last resolution level of the reference-grid in the $\{U,V,W\}$ directions. The acceptable values for {*nSlices.u*, *nSlices.v, nSlices.w*} and {*nROIs.u*, *nROIs.v*, *nROIs.w*} lie in the range [1, 263 -1]. Further it reads 6 flags defined in Table 42.

Table 42 — Meaning of the flags

Flag	Meaning
hasConnectivityInfo	Boolean flag: 1. When set to '1' it indicates that the parts identified by the MGMeshInfoTag, MGMeshConnectivityROInfoTag and MGMeshConnectivityInfoTag tags can be present in the stream at any resolution level description. 2. When set to '0' it implies that the parts identified by the MGMeshConnectivityROInfoTag and MGMeshConnectivityInfoTag tags are not present in the stream. If the value of the hasRefineInfo flag is '1', then the part identified by the MGMeshInfoTag tag can be present in the stream only for the first resolution level description. If the meshType parameter defined in Table 43 has the value '2' then a default quadrilateral mesh is generated as explained in subclause 4.2.3.3.5.
hasRefineInfo	Boolean flag: 1. When set to '1' it indicates that the parts identified by the MGMeshInfoTag, MGVerticesRefinementROIInfoTag and MGVerticesRefinementInfoTag tags can be present in the stream at any resolution level description. If the value of the hasConnectivityInfo flag is '0' these parts can only be available for the first resolution level description, as explained in subclause 4.2.3.3.5. 2. When set to '0' it implies that the parts identified by the MGVerticesRefinementROIInfoTag and MGVerticesRefinementInfoTag tags are not present in the stream. The part identified by the MGMeshInfoTag tag can be present in the stream at any resolution level description only if the value of the hasConnectivityInfo flag is set to '1'.
hasRepositionInfo	Boolean flag: 1. It can be set to '1' only if the hasConnectivityInfo flag is also '1'. When equal to '1' it indicates that the parts identified by the MGMeshInfoTag, MGVerticesRepositionROIInfoTag and MGVerticesRepositionInfoTag tags can be present in the stream at any resolution level description except the last resolution level. 2. When set to '0' it implies that the parts identified by the MGVerticesRepositionROIInfoTag and MGVerticesRepositionInfoTag tags are not present in the stream. The part identified by the MGMeshInfoTag tag can be present in the stream at any resolution level description only if the value of one of the following two flags is set to '1': hasConnectivityInfo, hasRefineInfo.
hasGridInfo	Boolean flag: 1. When set to '1' it indicates that the parts identified by the MGGridInfoTag, MGGridCoefficientsROIInfoTag and MGGridCoefficientsInfoTag tags can be present in the stream at any resolution level description. 2. When set to '0' it implies that the parts identified by the MGGridInfoTag, MGGridCoefficientsROIInfoTag and MGGridCoefficientsInfoTag tags are not present in the stream. In this case the reference-grid points are uniformly distributed and their coordinates are computed as a linear interpolation between the eight grid corners parsed by GridCorners in the MeshGridDecoderConfig class.
attributes	A 8-bit flag reserved for future use.

hasMultiLayer	Boolean flag: 1. If set to '1' it indicates that the model consists of several surface layers. The identifiers for the surface layers are encoded on nBitsLayer bits. 2. When set to '0' then either the model consists of one surface layer or no distinction between the surface layers is made.
uniformSplit	Boolean flag: 1. When set to '1' it indicates that the stream contains a quadrilateral mesh allowing to obtain the connectivity-wireframe for the higher resolution levels by uniformly splitting each quad recursively into four sub-quads, as illustrated in Figure 61. The parts identified by the MGMeshInfoTag, MGMeshConnectivityROInfoTag and MGMeshConnectivityInfoTag tags are present in the stream only at the first resolution level description. 2. When set to '0' the parts identified by the MGMeshInfoTag, MGMeshConnectivityROInfoTag and MGMeshConnectivityInfoTag tags can be present in the stream at any resolution level description.

Further the **MeshGridDecoderConfig** class parses **(5)** the type of the mesh (*meshType*) as explained in Table 43.

Table 43 — Encoding of the mesh type (*meshType*)

Value	Meaning
0	GENERIC_MESH: Non-homogeneous mesh that may consist of polygons ranging from triangles to heptagons.
1	TRI_MESH: Homogeneous triangular mesh
2	QUADRI_MESH: Homogeneous quadrilateral mesh
3	HEXA_MESH: Homogeneous hexagonal mesh

If *meshType* is quadrilateral (*QUADRI_MESH*) than **(6)** a 1-bit flag (*sameBorderOrientation*) defines the number of bits used to store the connectivity links between the vertices, as explained in subclause 4.2.3.3.1.1, and defined in Table 44, and **(7)** another 1-bit flag (*uniformSplit*) indicates the type of multiresolution as explained in subclause 4.2.3.3.5 and Table 42. If the number of slices (*nSlices*) in any *W* direction is equal to '1', i.e. the reference-grid consists of one layer of points, then **(8)** the variable length variable *offsetAmplitude* defines the maximum value of the offsets as explained in subclause 3.3.3.3.2.

Table 44 — Encoding of the sameBorderOrientation flag

Value	Meaning
0	2-bits are used to encode a connectivity link between two vertices; it is the general case.
1	1-bit is used to encode a connectivity link between two vertices; it is a particular case which may occur for homogeneous quadrilateral meshes.

Further the **MeshGridDecoderConfig** class parses **(9)** the *cyclic_folded* mode variable defined in Table 45 that specifies the cyclic behaviour of the mesh, as explained in subclause 4.2.3.3.1.3, **(10)** the description of

the vertices' refinement bits (*refine*), **(11)** the type of filter used in the wavelet transform (*filterType*), encoded on FILTER_TYPE bits, **(12)** the scaling factor for the {x,y,z} grid coefficients (*gridScale*), encoded on FIELD_BITS bits, **(13)** the {x,y,z} coordinates of the 8 corners of the reference-grid (*gridCorner*).

Table 45 — Encoding of the cyclic_folded mode variable

Bits	Value	Meaning
000	0	NON_CYCLIC: general case non-cyclic non-folded mesh.
001	1	CYCLIC_U: cyclic mesh in the "U" direction.
011	3	CYCLIC_UV: cyclic mesh in the "UV" direction.
101	5	FOLDED_SINGLE: cyclic mesh in the "U" direction and folded in the "V" direction at the end corresponding to the first index, i.e. $index = 0$. Notice that for the "V" direction the index varies between $index \in [0, nSlices[1] - 1]$.
111	7	FOLDED_BOTH: cyclic mesh in the "U" direction and folded in the "V" direction at both ends, i.e. $index = \{0, nSlices[1]\}$.
010	2	Reserved
100	4	Reserved
110	6	Reserved
Bits	**Value**	**Meaning**
000	0	NON_CYCLIC: general case non-cyclic non-folded mesh.
001	1	CYCLIC_U: cyclic mesh in the "U" direction.
011	3	CYCLIC_UV: cyclic mesh in the "UV" direction.
101	5	FOLDED_SINGLE: cyclic mesh in the "U" direction and folded in the "V" direction at the end corresponding to the first index, i.e. $index = 0$. Notice that for the "V" direction the index varies between $index \in [0, nSlices[1] - 1]$.
111	7	FOLDED_BOTH: cyclic mesh in the "U" direction and folded in the "V" direction at both ends, i.e. $index = \{0, nSlices[1]\}$.
010	2	Reserved
100	4	Reserved
110	6	Reserved

4.2.3.2.3 MeshGridStream

4.2.3.2.3.1 Syntax

```
aligned(8) expandable(2^28-1) class MeshGridStream
{
   MeshGridCommand[] commandUnits;
}
```

4.2.3.2.3.2 Semantics

The MeshGrid stream is an array of MeshGridCommand units.

4.2.3.2.4 MeshGridCommand

4.2.3.2.4.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class MeshGridCommand : bit(8) tag=0
{
   // empty. To be filled by classes extending this class.
}
```

4.2.3.2.4.2 Semantics

This is an abstract base class for the different types of command units of the MeshGrid stream. This class is extended by the classes identified by the class tags defined in Table 46.

Table 46 — MeshGrid command table

Tag value	Tag name	Description
0x00	Forbidden	
0x01	MGInfoTag	Tag for the MeshGrid stream coding information.
0x02	MGMeshInfoTag	Tag for the mesh coding information for a specified mesh resolution level.
0x03	MGGridInfoTag	Tag for grid coding information for a specified grid resolution level.
0x04	MGMeshConnectivityROIInfoTag	Tag for mesh connectivity information for a specified mesh resolution level and regions of interest (ROIs) list.
0x05	MGMeshConnectivityInfoTag	Tag for mesh connectivity information for a specified mesh resolution level.
0x06	MGVerticesRepositionROIInfoTag	Tag for vertices' reposition bits (single bit-plane) for a specified mesh resolution level and regions of interest (ROIs) list.
0x07	MGVerticesRepositionInfoTag	Tag for vertices' reposition bits (single bit-plane) for a specified mesh resolution level.
0x08	MGVerticesRefinementROIInfoTag	Tag for refinement bit-planes (the offset) for a specified mesh resolution level and regions of interest (ROIs) list.

0x09	MGVerticesRefinementInfoTag	Tag for refinement bit-planes (the offset) for a specified mesh resolution level.
0x10	MGGridCoefficientsROIInfoTag	Tag for wavelet coefficients for a specified grid resolution level and tiles list.
0x11	MGGridCoefficientsInfoTag	Tag for wavelet coefficients for a specified grid resolution level.
0x12	MGGridCornersInfoTag	Tag for the grid corners.
0x13-0xFE	Reserved for ISO use	
0xFF	Forbidden	

4.2.3.2.5 MGLevelDescriptor

4.2.3.2.5.1 Syntax

```
abstract class MGLevelDescriptor extends MeshGridCommand: bit(8) tag=0
{
   // read the variable length counter sizeOfInstance
   unsigned int(LEVEL_BITS) resolutionLevel;
   bit(4) flags;
}
```

4.2.3.2.5.2 Semantics

This is an abstract class that serves as a base class for the MeshGrid stream unit classes. MGLevelDescriptor reads **(1)** the resolution level (*resolutionLevel*) of the MeshGrid stream unit, and **(2)** a flag (*flags*) (defined in Table 11) that specifies how to handle the decoded MeshGrid stream unit. An example (exploiting the functionality provided by the *flags* field) showing the morphing of a MeshGrid model allowing for topological changes is displayed in Figure 46.

Table 47 — Values of flags

Bits	Meaning
0000	If first bit is '0' then the received data is an update to existing data.
0001	If first bit is '1' then the received data replaces any existing data.
	Remaining 3 bits are reserved.

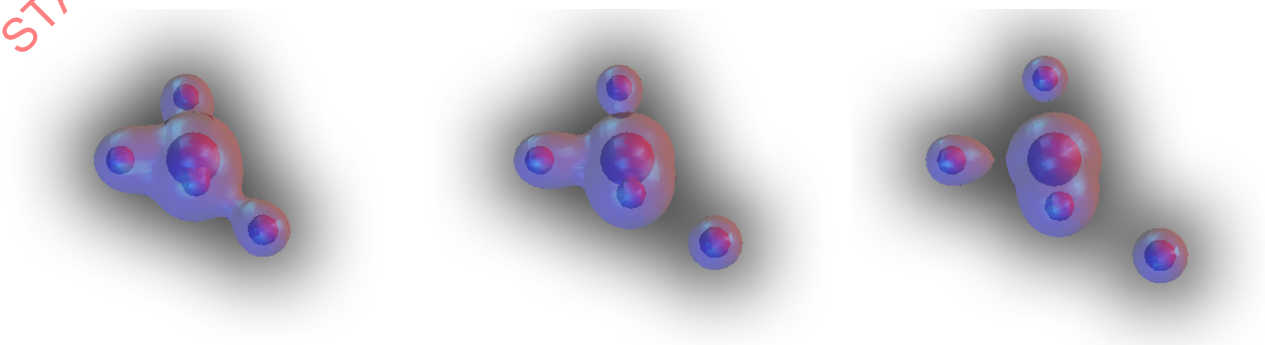


Figure 46 — Exploiting the functionality provided by the flags field for efficient morphing.

4.2.3.2.6 MGMeshDescriptor

4.2.3.2.6.1 Syntax

```
class MGMeshDescriptor(MeshGridDecoderConfig mgd) extends MGLayerDescriptor: bit(8)
tag=MGMeshInfoTag
{
bool bLastLevel = resolutionLevel.value == mgd.totalNumLevelsMesh - 1;
bool bReposition = mgd.hasRepositionInfo && !bLastLevel;
   bool bRefine = mgd.hasRefineInfo && (mgd.refine.bFull || bLastLevel);
   MeshDescriptor mdl[[resolutionLevel.value]](mgd.hasConnectivityInfo, bReposition,
bRefine);
}
```

4.2.3.2.6.2 Semantics

The MGMeshDescriptor class parses the mesh coding information for a specified mesh resolution level and layers.

4.2.3.2.7 MGLayerDescriptor

4.2.3.2.7.1 Syntax

```
abstract class MGLayerDescriptor extends MGLevelDescriptor(MeshGridDecoderConfig mgd):
bit(8) tag=0
{
   // read the variable length counter sizeOfInstance
   if (mgd.hasMultiLayer)
      ParseIndices identifier(mgd.nBitsLayer);
}
```

4.2.3.2.7.2 Semantics

This is an abstract class that serves as a base class for the MeshGrid stream unit classes. MGLayerDescriptor reads an *identifier* of the surface layer the MeshGrid stream unit refers to. When the value of *identifier.number* is '0' then the contents of the MeshGrid stream unit is generic for all surface layers. Figure 48 illustrates an example of a MeshGrid model consisting of several surface layers.

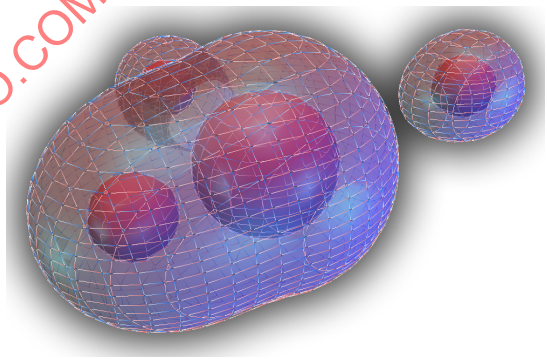


Figure 47 — MeshGrid model consisting of several surface layers.

4.2.3.2.8 MGGridCorners

4.2.3.2.8.1 Syntax

```
class MGGridCorners extends MeshGridCommand(MeshGridDecoderConfig mgd): bit(8)
tag=MGGridCornersInfoTag
{
   // read the coordinates of the grid corners
```

```
   GridCorners gridCorner(mgd.nSlices, mgd.cyclic_folded);
}
```

4.2.3.2.8.2 Semantics

The MGGridCorners class parses the grid corners (gridCorner) as explained in subclause 4.2.3.2.24.

4.2.3.2.9 MGGridDescriptor

4.2.3.2.9.1 Syntax

```
class MGGridDescriptor extends MGLevelDescriptor: bit(8) tag=MGGridInfoTag
{
   GridDescriptor gdl[[resolutionLevel.value]]();
}
```

4.2.3.2.9.2 Semantics

The MGGridDescriptor class parses the grid coding information for a specified grid resolution level.

4.2.3.2.10 MGMeshConnectivityROIDescriptor

4.2.3.2.10.1 Syntax

```
class MGMeshConnectivityROIDescriptor(MeshDescriptor mdl) extends MGLayerDescriptor:
bit(8) tag=MGMeshConnectivityROIInfoTag
{
   ParseIndices roi(mdl[resolutionLevel.value].nBitsIndex);
   MeshConnectivity
mc[[resolutionLevel.value]](mdl[resolutionLevel.value].nBitsConnectivity,roi.number,
roi.index);
}
```

4.2.3.2.10.2 Semantics

This class parses the mesh connectivity bits for specified layers, regions of interest (roi) and mesh resolution level (resolutionLevel).

4.2.3.2.11 MGMeshConnectivityDescriptor

4.2.3.2.11.1 Syntax

```
class MGMeshConnectivityDescriptor(MeshDescriptor mdl) extends MGLevelDescriptor: bit(8)
tag=MGMeshConnectivityInfoTag
{
   MakeIndices index(mdl[resolutionLevel.value].nROIs);
   MeshConnectivity mc[[resolutionLevel.value]]
(mdl[resolutionLevel.value].nBitsConnectivity,mdl[resolutionLevel.value].nROIs,
index.value);
}
```

4.2.3.2.11.2 Semantics

This class parses the mesh connectivity bits for all layers, all regions of interest (identified by their indices (index)), and mesh resolution level (resolutionLevel).

4.2.3.2.12 MGVerticesRepositionROIDescriptor

4.2.3.2.12.1 Syntax

```
class MGVerticesRepositionROIDescriptor(MeshDescriptor mdl, MeshConnectivity mc) extends
MGLayerDescriptor: bit(8) tag=MGVerticesRepositionROIInfoTag
{
   ParseIndices roi(mdl[resolutionLevel.value].nBitsIndex);
   VertexRefine vrep[[resolutionLevel.value]](mdl[resolutionLevel.value].nBitsReposition,
                                              roi.number, roi.index,
                                              mc[resolutionLevel.value].bMeshPresent);
}
```

4.2.3.2.12.2 Semantics

This class parses the vertices' reposition bit-plane for specified layers, regions of interest (roi) and mesh resolution level (resolutionLevel).

4.2.3.2.13 MGVerticesRepositionDescriptor

4.2.3.2.13.1 Syntax

```
class MGVerticesRepositionDescriptor(MeshDescriptor mdl, MeshConnectivity mc) extends
MGLevelDescriptor: bit(8) tag=MGVerticesRepositionInfoTag
{
   MakeIndices index(mdl[resolutionLevel.value].nROIs);
   VertexRefine vrep[[resolutionLevel.value]](mdl[resolutionLevel.value].nBitsReposition,
                                    mdl[resolutionLevel.value].nROIs, index.value,
                                    mc[resolutionLevel.value].bMeshPresent);
}
```

4.2.3.2.13.2 Semantics

This class parses the vertices' reposition bit-plane for all layers, all regions of interest (identified by their indices (index)), and mesh resolution level (resolutionLevel).

4.2.3.2.14 MGVerticesRefinementROIDescriptor

4.2.3.2.14.1 Syntax

```
class MGVerticesRefinementROIDescriptor(MeshDescriptor mdl, MeshConnectivity mc) extends
MGLayerDescriptor: bit(8) tag=MGVerticesRefinementROIInfoTag
{
   unsigned int(REFINE_BITS) startBitPlane;
   unsigned int(REFINE_BITS) endBitPlane;
   ParseIndices roi(mdl[resolutionLevel.value].nBitsIndex);
   VertexRefine vref[[resolutionLevel.value]](mdl[resolutionLevel.value].nBitsRefine,
                                              roi.number, roi.index,
                                              mc[resolutionLevel.value].bMeshPresent);
}
```

4.2.3.2.14.2 Semantics

This class parses the vertices' refinement bit-planes, starting with startBitPlane and ending with endBitPlane, for specified layers, regions of interest (roi) and mesh resolution level (resolutionLevel).

4.2.3.2.15 MGVerticesRefinementDescriptor

4.2.3.2.15.1 Syntax

```
class MGVerticesRefinementDescriptor(MeshDescriptor mdl, MeshConnectivity mc) extends
MGLevelDescriptor: bit(8) tag=MGVerticesRefinementInfoTag
{
   MakeIndices index(mdl[resolutionLevel.value].nROIs);
   VertexRefine vref[[resolutionLevel.value]](mdl[resolutionLevel.value].nBitsRefine,
                                    mdl[resolutionLevel.value].nROIs, index.value,
                                    mc[resolutionLevel.value].bMeshPresent);
}
```

4.2.3.2.15.2 Semantics

This class parses all the vertices' refinement bit-planes for all layers, all regions of interest (identified by their indices (index)), and mesh resolution level (resolutionLevel).

4.2.3.2.16 MGGridCoefficientsROIDescriptor

4.2.3.2.16.1 Syntax

```
class MGGridCoefficientsROIDescriptor(GridDescriptor gdl) extends MGLevelDescriptor:
bit(8) tag=MGGridCoefficientsROIInfoTag
{
   unsigned int(FIELD_BITS) startBitPlane;
   ParseIndices tile(gdl[resolutionLevel.value].nBitsIndex);
   GridCoefficients gc[[resolutionLevel.value]] (gdl[resolutionLevel.value].counter,
                                                 gdl[resolutionLevel.value].nBitsCounter,
                                                 tile.number, tile.index);
}
```

4.2.3.2.16.2 Semantics

This class parses parts of the component streams, i.e. three binary streams each of them corresponding to one of the {x,y,z} wavelet encoded coordinates of the grid points, for specified tiles (tile) and grid resolution level (resolutionLevel). The start bitplane (startBitPlane) is specified and it is the same for each component stream. The number of bytes received for each component stream and the corresponding wavelet coefficients are parsed by the GridCoefficients class.

4.2.3.2.17 MGGridCoefficientsDescriptor

4.2.3.2.17.1 Syntax

```
class MGGridCoefficientsDescriptor(GridDescriptor gdl) extends MGLevelDescriptor: bit(8)
tag=MGGridCoefficientsInfoTag
{
   MakeIndices index(gdl[resolutionLevel.value].totalNumTiles);
   GridCoefficients gc[[resolutionLevel.value]] (gdl[resolutionLevel.value].counter,
                                                 gdl[resolutionLevel.value].nBitsCounter,
                                                 gdl[resolutionLevel.value].totalNumTiles,
                                                 index.value);
}
```

4.2.3.2.17.2 Semantics

This class parses parts of the component streams, i.e. three binary streams each of them corresponding to one of the {x,y,z} wavelet encoded coordinates of the grid points, for all the tiles,identified by their indices (index),and grid resolution level (resolutionLevel). The number of bytes received for each component stream and the corresponding wavelet coefficients are parsed by the GridCoefficients class.

4.2.3.2.18 MeshDescriptor

4.2.3.2.18.1 Syntax

```
// Mesh Description for level
aligned(8) class MeshDescriptor(bool hasConnectivity, bool hasReposition, bool hasRefine)
{
   // some member variables
   unsigned int nBitsIndex, totalNumROIs;

      // compute the number of slices and number of ROIs at resolutionLevel
(MGLevelDescriptor)
   // from nROIs, nLevels and nSlices specified in MeshGridDecoderConfig (mgd)
   ComputeNrSlices nSlices(mgd.nSlices, mgd.nLevels, resolutionLevel);
   ComputeNrROIs nROIs(mgd.nROIs, mgd.nLevels, resolutionLevel);

 if (hasReposition) {
     // retrieve the reposition bits flag
     bit (2) repositionBits;
 }

   if (hasConnectivity) {
      // number of bits for the ROI indices fields
      nBitsIndex = (int) floor(log2(nROIs.totalNumROIs) + 1);
      // number of bits for the counter fields
      unsigned int(VERTS_BITS) nBitsConnectivity;
   }

   if (hasReposition && repositionBits == READ_REPOSITION_BITS) {
      unsigned int(VERTS_BITS) nBitsReposition;
   }

   if (hasRefine) {
      unsigned int(VERTS_BITS) nBitsRefine;
   }
}
```

4.2.3.2.18.2 Semantics

The **MeshDescriptor** class parses the coding information for a specified mesh resolution level. It first computes **(1)** the number of slices (*nSlices*) and **(2)** the number of ROIs (*nROIs*) for each {*u,v,w*} direction. Further, if the *hasReposition* flag is set to '1' (the *hasRepositionInfo* flag from the *MeshGridDecoderConfig* is set to '1' and the specified mesh resolution level is not the last) it reads **(3)** a 2-bit flag (*repositionBits*), which indicates the default value of the vertices' reposition bits or their presence in the stream as given in Table 48. When a default value is specified, the reposition bits are not present in the stream for the specified resolution level.

Table 48 — Encoding of the repositionBits flag

Value	Meaning
0	No reposition bits encoded, all have default value 0.
1	No reposition bits encoded, all have default value 1.
2	No default value, the reposition bits are present in the stream.
3	Reserved

Further, if the *hasConnectivity* flag is set to '1' (the *hasConnectivityInfo* flag from the *MeshGridDecoderConfig* is set to '1') the **MeshDescriptor** class **(4)** computes the number of bits (*nBitsIndex*) in which the indices of the ROIs are stored, and **(5)** parses the number of bits (*nBitsConnectivity*) used for storing the length (in

bytes) of the coded mesh connectivity. If the *hasReposition* flag is set to '1' and no default value is specified for the vertices' reposition bits (*repositionBits*), then **(6)** the number of bits (*nBitsReposition*) used for storing the length (in bytes) of the coded vertices' reposition information is retrieved. If the *hasRefine* flag is '1' (the *hasRefineInfo* flag from the *MeshGridDecoderConfig* is set to '1' and it is the last resolution level or the full refine flag has been specified), then **(7)** the number of bits (*nBitsRefine*) used for storing the length (in bytes) of the coded vertices' refinement information is parsed.

4.2.3.2.19 GridDescriptor

4.2.3.2.19.1 Syntax

```
// Grid Description for level
aligned(8) class GridDescriptor() {
   // some member variables
   unsigned int nBitsIndex, totalNumTiles;

   // compute the number of slices and Tiles at resolutionLevel (MGLevelDescriptor) from
   // nROIs, nLevels and nSlices specified in MeshGridDecoderConfig (mgd)
   ComputeNrSlices nSlices(mgd.nSlices, mgd.nLevels, resolutionLevel);
   ComputeNrROIs nTiles(mgd.nROIs, mgd.nLevels, resolutionLevel);

   // number of bits for the ROI indices fields
   nBitsIndex = (int) floor(log2(nTiles.totalNumROIs) + 1);

   // the threshold
   Threshold threshold;

   // allocate an array keeping the number of bytes per Tile
   // it will be initialized in GridCoefficients
   PointXYZ counter[nTiles.totalNumROIs];
   for (i = 0; i < nTiles.totalNumROIs; i++) {
      counter[i] = {0,0,0};
   }

   // number of bits for the counter fields (it can be 0)
   ParsableXYZ nBitsCounter(GRID_BITS);
}
```

4.2.3.2.19.2 Semantics

The **GridDescriptor** class parses the grid header for a given resolution level. It first computes **(1)** the number of slices (*nSlices*) and **(2)** the number of tiles (*nTiles*) for each {*u,v,w*} direction. Further, it **(3)** parses the threshold (*threshold*) for each {x,y,z} coded grid tile, **(4)** computes the number of bits needed for specifying the tile indices, **(5)** allocates the array which stores the number of bytes of coefficients received per tile, and **(6)** parses a GRID_BITS bits value (*nBitsCounter*), representing the number of bits allocated for the counters storing the length (in bytes) for each {x,y,z} coded grid tile.

4.2.3.2.20 MeshConnectivity

4.2.3.2.20.1 Syntax

```
class MeshConnectivity (unsigned int nBitsCounter, unsigned int numberOfROIs, unsigned int
index[])
{
   unsigned int i = 0;

   // get ROIs lookup
   for (i = 0; i < numberOfROIs; i++) {
      bit(1) bMeshPresent[[index[i]]];
   }

   // get the counters for the ROIs
   for (i = 0; i < numberOfROIs; i++) {
```

```
      int counter[[index[i]]] = 0;
      if (bMeshPresent[index[i]] == 1) {
         int (nBitsCounter) counter[[index[i]]];
      }
   }

   // get the ROI coding bits
   for (i = 0; i < numberOfROIs; i++) {
      unsigned int count;
      for (count = 0; count < counter[index[i]]; count++) {
         unsigned int(8) data;
      }
   }
}
```

4.2.3.2.20.2 Semantics

The **MeshConnectivity** class parses the mesh connectivity data for a set of ROIs at a specified resolution level. It reads, for each ROI (identified by the *index*), a 1-bit flag (*bMeshPresent*), which, if true, means that the mesh is passing through the ROI. Further, if *bMeshPresent* is true the **MeshConnectivity** class reads *nBitsCounter* bits representing the length (in bytes), specified by the *counter*, of the coded connectivity information. The connectivity information is stored in buffer *data*. The decoding of the connectivity information is explained in subclause 4.2.3.3.1.

4.2.3.2.21 VertexRefine

4.2.3.2.21.1 Syntax

```
class VertexRefine (unsigned int nBitsCounter, unsigned int numberOfROIs, unsigned int
index[], bit bMeshPresent[])
{
   unsigned int i = 0;

   // get the counters for the ROIs
   for (i = 0; i < numberOfROIs; i++) {
      int counter[[index[i]]] = 0;
      if (bMeshPresent[index[i]] == 1) {
         int (nBitsCounter) counter[[index[i]]];
      }
   }

   // get the vertices' reposition/refine bits
   for (i = 0; i < numberOfROIs; i++) {
      unsigned int count;
      for (count = 0; count < counter[index[i]]; count++) {
         unsigned int(8) data;
      }
   }
}
```

4.2.3.2.21.2 Semantics

The **VertexRefine** class parses the vertices refine/reposition data for a set of ROIs at a specified resolution level. It checks if the mesh is present in the ROI (identified by the *index*) by testing the *bMeshPresent* flag, which is initialized by the *MeshConnectivity* class. Further, if the *bMeshPresent* evaluates to true, the **VertexRefine** class reads nBitsCounter bits representing the length (in bytes) of the coded reposition/refinement information. The reposition/refinement information is stored in buffer *data*. The decoding of the reposition/refinement information is explained in subclause 4.2.3.3.3.

4.2.3.2.22 GridCoefficients

4.2.3.2.22.1 Syntax

```
class GridCoefficients(PointXYZ counter[], ParsableXYZ nBitsGrid, unsigned int
numberOfTiles, unsigned int index[])
{
   unsigned int i = 0;

   // get the counters for the tiles and update the total counters
   // in case the grid coefficients are read progressively
   for (i = 0; i < numberOfTiles; i++) {
      CounterXYZ frameCounter[[index[i]]](nBitsGrid);
      counter[index[i]] += frameCounter[index[i]];
   }

   // read the coefficients and append them to the existing ones
   // in case the grid coefficients are read progressively
   for (i = 0; i < numberOfTiles; i++) {
      unsigned int count;

      // read the coded coefficients of the 'x' coordinates
      for (count = 0; count < frameCounter[index[i]].x; count++)
         unsigned int(8) dataX;

      // read the coded coefficients of the 'y' coordinates
      for (count = 0; count < frameCounter[index[i]].y; count++)
         unsigned int(8) dataY;

      // read the coded coefficients of the 'z' coordinates
      for (count = 0; count < frameCounter[index[i]].z; count++)
         unsigned int(8) dataZ;
   }
}
```

4.2.3.2.22.2 Semantics

The **GridCoefficients** class parses the grid coefficients data for a set of ROIs at a specified resolution level. It reads for each tile (identified by the *index*) the length (in bytes), specified in *frameCounter* for each coordinate {x,y,z}, from the coded grid information. The decoding of the grid information is explained in subclause 4.2.3.3.2. The grid coefficients are stored in the buffers *dataX*, *dataY*, *dataZ*.

4.2.3.2.23 RefineVertexDescriptor

4.2.3.2.23.1 Syntax

```
// number of refine bits vertex
class RefineVertexDescriptor {
   if (hasRefineInfo) {
      // full refine at each level
      bit(1)  bFull;

      // number of refine bits should be larger than 0
      // otherwise the hasRefineInfo flag should be 0
      unsigned int(REFINE_BITS) nBits;
   }
   else {
      unsigned int bFull = 0;
      unsigned int nBits = 0;
   }
}
```

4.2.3.2.23.2 Semantics

The **RefineVertexDescriptor** class reads **(1)** the 1-bit flag (*bFull*) indicating the presence of vertices' refinement information coded for each resolution level of the mesh, and **(2)** REFINE_BITS bits value representing the number of quantization bits of the vertices' offsets (*nBits*).

4.2.3.2.24 GridCorners

4.2.3.2.24.1 Syntax

```
class GridCorners(PointUVW nSlices, BYTE cyclic_folded) {
  unsigned int(QUANT_BITS) nBits;

   // check for single layer and cyclic/folded
   unsigned int numCorners = 8;
   if (nSlices.w == 1)
      numCorners /= 2;
   if (cyclic_folded == CYCLIC_U)
      numCorners /= 2;
   else if (cyclic_folded == CYCLIC_UV)
      numCorners /= 4;

   unsigned int idx;
   PointXYZ value[numCorners];
   for (idx = 0; idx < numCorners; idx ++) {
      // the x coordinate
      bit(1) sgn;                  // the sign
      unsigned int(nBits) tmp;     // the value

      if (sgn == 1)   value[idx].x = -tmp;
      else            value[idx].x =  tmp;

      // the y coordinate
      bit(1) sgn;                  // the sign
      unsigned int(nBits) tmp;     // the value

      if (sgn == 1)   value[idx].y = -tmp;
      else            value[idx].y =  tmp;

      // the z coordinate
      bit(1) sgn;                  // the sign
      unsigned int(nBits) tmp;     // the value

      if (sgn == 1)   value[idx].z = -tmp;
      else            value[idx].z =  tmp;
   }
}
```

4.2.3.2.24.2 Semantics

The **GridCorners** class retrieves the coordinates {x,y,z} of the eight corners of the grid. It reads **(1)** a QUANT_BITS bits value (*nBits*) indicating the number of bits allocated for the *value* fields. For each of the {x,y,z} coordinates of the corners**,** it reads **(2)** a 1-bit *sgn*, which, when equal to '1', indicates that the following value is negative, and **(3)** a *nBit* value (*value*), representing the absolute coordinate value.

4.2.3.2.25 ScaleXYZ

4.2.3.2.25.1 Syntax

```
// group the x,y,z directions
class ScaleXYZ(unsigned int count) {
  // count and nBits should be larger than 0
  unsigned int(count) nBits;
```

```
  // the actual fields
  unsigned int(nBits) x;
  unsigned int(nBits) y;
  unsigned int(nBits) z;
}
```

4.2.3.2.25.2 Semantics

The **ScaleXYZ** class parses the scaling factors for decoding the wavelet coefficients of the {x,y,z} coordinates of the reference-grid points. It reads *nBits* bits for each coordinate {*x,y,z*}. The scaling values are used to scale down the decoded wavelet coefficients by means of integer division, before applying the wavelet reconstruction (see subclause 4.2.3.3.2.1) of the reference-grid coordinates.

4.2.3.2.26 PointXYZ

4.2.3.2.26.1 Syntax

```
class PointXYZ {
  // the actual fields
  int x, y, z;
}
```

4.2.3.2.26.2 Semantics

The **PointXYZ** class is a structure grouping the coordinates {x,y,z} of the reference-grid points. It does not read values from the stream.

4.2.3.2.27 ParsableUVW

4.2.3.2.27.1 Syntax

```
class ParsableUVW() {
   unsigned int u = ParseValue();
   unsigned int v = ParseValue();
   unsigned int w = ParseValue();
}
```

4.2.3.2.27.2 Semantics

The **ParsableUVW** class reads three variables $\{u,v,w\}$ that are related to the $\{U,V,W\}$ directions of the reference-grid.

4.2.3.2.28 PointUVW

4.2.3.2.28.1 Syntax

```
class PointUVW {
  // the actual fields
  unsigned int u, v, w;
}
```

4.2.3.2.28.2 Semantics

The PointUVW class is a structure grouping the positions $\{u,v,w\}$ in the reference-grid. It does not read values from the stream.

4.2.3.2.29 FixedUVW

4.2.3.2.29.1 Syntax

```
// group the u,v,w directions
class FixedUVW (unsigned int nBits) {
  unsigned int(nBits) u;
  unsigned int(nBits) v;
  unsigned int(nBits) w;
}
```

4.2.3.2.29.2 Semantics

The **FixedUVW** class reads three variables $\{u,v,w\}$ that are related to the $\{U,V,W\}$ directions of the reference-grid.

4.2.3.2.30 ParsableXYZ

4.2.3.2.30.1 Syntax

```
class ParsableXYZ(unsigned int nBits) {
   unsigned int(nBits) x;
   unsigned int(nBits) y;
   unsigned int(nBits) z;
}
```

4.2.3.2.30.2 Semantics

The **ParsableXYZ** class reads three variables $\{x,y,z\}$ that are related to the coordinates of the reference-grid.

4.2.3.2.31 CounterXYZ

4.2.3.2.31.1 Syntax

```
// group the x,y,z coordinates
class CounterXYZ(ParsableXYZ nBits) {
  // the actual fields
  unsigned int x, y, z;
  x = y = z = 0;

  // number of bits allocated for the x,y,z fields; it can be '0'
  if (nBits.x > 0) {
    // coding of the x coordinate
    bit(1) bGridCoded;
    if (bGridCoded == 1) unsigned int(nBits.x) x;
  }

  if (nBits.y > 0) {
    // coding of the y coordinate
    bit(1) bGridCoded;
    if (bGridCoded == 1) unsigned int(nBits.y) y;
  }

  if (nBits.z > 0) {
    // coding of the z coordinate
    bit(1) bGridCoded;
    if (bGridCoded == 1) unsigned int(nBits.z) z;
  }
}
```

4.2.3.2.31.2 Semantics

The **CounterXYZ** class reads the counters storing the length (in bytes) of the coded grid tiles for each of the {x,y,z} grid coordinates. If the *nBits* value is equal to '0', the counters {*x,y,z*} are set to '0'; otherwise, for each coordinate, the CounterXYZ class reads **(1)** the 1-bit flag (*bGridCoded*), indicating, when '0', that no grid coefficients have been coded, thus no counter ({*x,y,z*}) is present; otherwise, the class retrieves the **(2)** *nBits* bits counters ({*x,y,z*}) storing the length (in bytes) of the coded grid tiles.

4.2.3.2.32 Threshold

4.2.3.2.32.1 Syntax

```
class Threshold {
  // the actual fields
  unsigned int x, y, z;
  x = y = z = 0;

  // the threshold for the coded x coordinate
  unsigned int(FIELD_BITS) t;
  if (t != 0)   x = 1 << (t - 1);

  // the threshold for the coded y coordinate
  unsigned int(FIELD_BITS) t;
  if (t != 0)   y = 1 << (t - 1);

  // the threshold for the coded z coordinate
  unsigned int(FIELD_BITS) t;
  if (t != 0)   z = 1 << (t - 1);
}
```

4.2.3.2.32.2 Semantics

For each of the {x,y,z} coded coordinates, the **Threshold** class reads, for each sub-band, the $\log_2$ of the threshold value (*t*). If *t* is equal to '0', the corresponding threshold {*x,y,z*} is '0'. Otherwise, the threshold is computed as $2^{(t-1)}$ (see subclause 4.2.3.3.2.2).

4.2.3.2.33 ParseIndices

4.2.3.2.33.1 Syntax

```
class ParseIndices(int nBitsIndex)
{
   unsigned int i = 0, j = 0, k = 0;

   // the number of elements in the indices list.
   unsigned int (nBitsIndex) number;

   for (i = 0; i < number; i++) {
   bit (1) isIndexed;
      if (isIndexed) {
         unsigned int (nBitsIndex) index[[k++]]; // index of the unit
      } else {
         unsigned int (nBitsIndex) start;        // start index of the unit
         unsigned int (nBitsIndex) count;        // number of indices in the range
         for (j = 0; j < count; j++) {
         index[[k++]] = start + j;
         }
      }
   }

    // assign the total number of units.
   number = k;
}
```

4.2.3.2.33.2 Semantics

The **ParseIndices** class retrieves a list of units that can be independently read, given a set of explicitly defined indices, and/or units that can be successively read, given a range of consecutive indices. First, it reads the number of elements in the indices list (*number*). A list element is the index of a unit (*index*), if the flag *isIndexed* is set to '1'; otherwise, the **ParseIndices** class retrieves the start index (*start*) of a range of indices, followed by a number (*count*) specifying the number of indices in the range. The indices are stored in *index* array.

4.2.3.2.34 ParseValue

4.2.3.2.34.1 Syntax

```
class ParseValue
{
   unsigned int length = 0;
   unsigned int number = 0;

   do {
      bit(1) bNextByte;
      number <<= 7;
      bit(7) value;
      number |= value;
      length += 8;
   }
   while (bNextByte);
}
```

4.2.3.2.34.2 Semantics

The **ParseValue** class parses a variable length integer. The result is returned in *number* and the number of bytes read in *length*.

4.2.3.2.35 MakeIndices

4.2.3.2.35.1 Syntax

```
class MakeIndices (int number)
{
   unsigned int i = 0;
   unsigned int value[number];

   for (i = 0; i < number; i++) {
   value[i] = i;
   }
}
```

4.2.3.2.35.2 Semantics

The **MakeIndices** class creates a list of consecutive indices. There is no stream access in this class.

4.2.3.2.36 MGDescriptor

4.2.3.2.36.1 Syntax

```
class MGDescriptor extends MeshGridCommand: bit(8) tag=MGInfoTag
{
   // read the variable length counter sizeOfInstance
   MeshGridDecoderConfig decoderConfig;
}
```

4.2.3.2.36.2 Semantics

The MGDescriptor class parses the coding information of the MeshGrid stream. This part is mandatory to be present at the beginning of the stream when the MeshGrid stream is carried in the buffer field of the Bitwrapper node during the in-band scenario, as explained in ISO/IEC 14496-11.

4.2.3.2.37 ComputeNrROIs

4.2.3.2.37.1 Syntax

```
class ComputeNrROIs (PointUVW nMaxROIs, PointUVW nLevels, unsigned int resolutionLevel)
{
   // compute the total number of resolution levels (totalNumLevels) and
   // for each {U,V,W} direction find the difference between totalNumLevels
   // and current resolution level (resolutionLevel)
   unsigned int totalNumLevels = max(nLevels.u, max(nLevels.v, nLevels.w));
   PointUVW levelDiff = min(totalNumLevels - resolutionLevel - 1, nLevels - 1);

   // compute the nROIs in the {U,V,W} directions
   unsigned int u = DevideByTwoEven(nMaxROIs.u, levelDiff.u);
   unsigned int v = DevideByTwoEven(nMaxROIs.v, levelDiff.v);
   unsigned int w = DevideByTwoEven(nMaxROIs.w, levelDiff.w);

   unsigned int totalNumROIs = u * v * w;
}
```

4.2.3.2.37.2 Semantics

The **ComputeNrROIs** class computes the number of ROIs of a lower resolution level given the number of ROIs (*nROIs*) at a higher resolution level and the level difference (*levelDiff*). There is no stream access in this class.

4.2.3.2.38 ComputeNrSlices

4.2.3.2.38.1 Syntax

```
class ComputeNrSlices (PointUVW nSlices, PointUVW nLevels, unsigned int resolutionLevel)
{
   // compute the total number of resolution levels (totalNumLevels) and
   // for each {U,V,W} direction find the difference between totalNumLevels
   // and current resolution level (resolutionLevel)
   unsigned int totalNumLevels = max(nLevels.u, max(nLevels.v, nLevels.w));
   PointUVW levelDiff = min(totalNumLevels - resolutionLevel - 1, nLevels - 1);

   unsigned int u = DevideByTwoOdd(nSlices.u, levelDiff.u);
   unsigned int u = DevideByTwoOdd(nSlices.v, levelDiff.v);
   unsigned int w = DevideByTwoOdd(nSlices.w, levelDiff.w);
}
```

4.2.3.2.38.2 Semantics

The **ComputeNrSlices** class computes the number of slices of a lower resolution level given the number of slices (nSlices) at a higher resolution level and the level difference (levelDiff). There is no stream access in this class.

4.2.3.2.39 DevideByTwoEven

4.2.3.2.39.1 Syntax

```
class DevideByTwoEven (int number, int times)
{
```

```
   unsigned int val = (number >> times);
}
```

4.2.3.2.39.2 Semantics

The **DevideByTwoEven** class performs the division of an even *number* by two to power *times*. There is no stream access in this class.

4.2.3.2.40 DevideByTwoOdd

4.2.3.2.40.1 Syntax

```
class DevideByTwoOdd (int number, int times)
{
   unsigned int val = ((number + (1 << times) - 1) >> times);
}
```

4.2.3.2.40.2 Semantics

The **DevideByTwoOdd** class performs the division of an odd *number* by two to power *times*. There is no stream access in this class.

4.2.3.3 Decoder

The *connectivity-wireframe* is coded using a new type of 3D extension of Freeman chain-code. The *reference-grid* is a vector field defined on a regular discrete 3D space, and the coordinates ($x(u,v,w)$,$y(u,v,w)$,$z(u,v,w)$), are compressed using an embedded 3D wavelet-based multi-resolution intra-band coding algorithm.

The decoder of the MeshGrid bitstream consists of three parts: (1) the connectivity-wireframe decoder, (2) the reference-grid decoder, and (3) the vertices' refinement decoder.

4.2.3.3.1 Connectivity-Wireframe Decoder

The connectivity-wireframe description is stored in the ***vertexLink*** field of the **MeshGrid** node (see subclause 3.3.3.2.2), and defines the connectivity vectors, between the vertices of the mesh as illustrated in Figure 48 for one reference-surface through a 3D object, and for the entire 3D object in Figure 49. A connectivity vector (label 7) has three constraints: (1) given a starting vertex V_P, a connectivity vector -L- from V_P will be located inside one of the reference-surfaces S_1 or S_2 passing through V_P, (2) a connectivity-vector will connect two vertices V_P and V_N that are lying the closest to each other at the same side of the object's surface, (3) the orientation of a connectivity vector inside reference-surface S, e.g. from vertex V_P to V_N, is defined by a counter-clockwise (*CCW*) or a clockwise (*CW*) scanning direction around a central point inside the object. The *CCW* scanning direction is imposed when the vertices V_P and V_N linked by the connectivity vector, are located on the external surface of the object. Respectively, the *CW* scanning direction is imposed when the vertices V_P and V_N are located on the internal surface of the object.

There will be two connectivity-vectors (-L_{N1}- and -L_{N2}-) going from vertex V to other vertices (*outgoing*), one connectivity vector being located inside reference-surface S_1, while the other one is located inside reference-surface S_2. Similarly, vertex V will be referred to by two *incoming* links (-L_{P1}- and -L_{P2}-) from two other vertices. The 4 neighboring vertices with V are named P_1, P_2, N_1 and N_2 (see Figure 49), and are connected with V via -L_{P1}-, -L_{P2}-, -L_{N1}- and -L_{N2}-. Vertex N_1 (respectively N_2) follows vertex V on curve C_1 (respectively C_2), and vertex P_1 (respectively P_2) precedes vertex V in curve C_1 (respectively C_2) for the imposed scanning orientation.

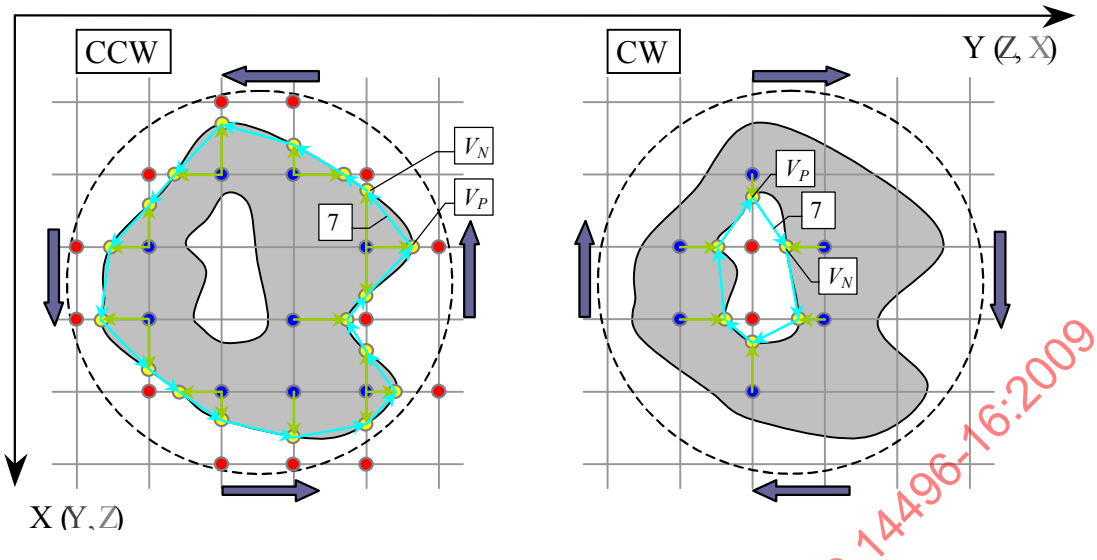


Figure 48 — A cross-section through a 3D object, illustrating the scanning direction, which is CCW for an external surface and CW for an internal surface. The connectivity vectors have the orientation of the scanning direction.

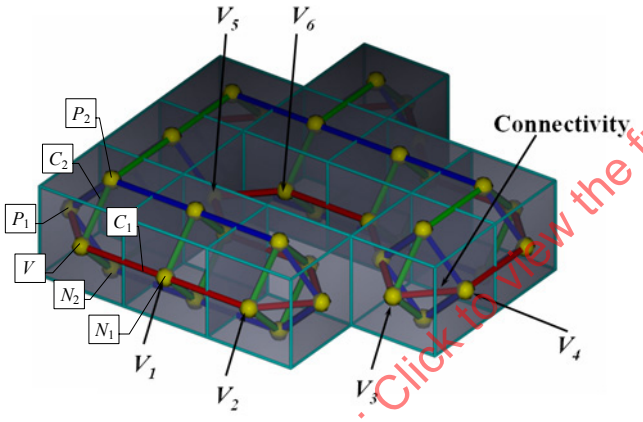


Figure 49 — The connectivity-wireframe of a discrete object, and the corresponding border voxels. It illustrates the relative positions between two connected vertices.

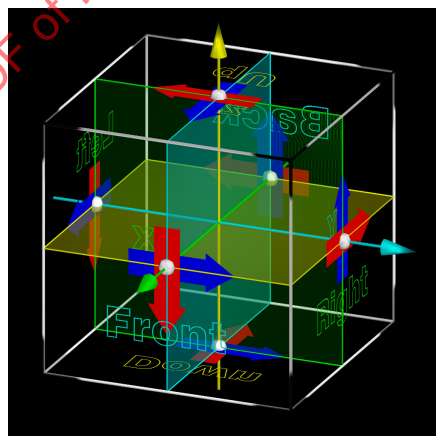


Figure 50 — The 6 discrete orientations of the faces of the border voxel, labeled as: Front, Back, Left, Right, Up, Down.

The indices {1, 2} of the connectivity vectors (-L_1-) and (-L_2-) are chosen in such a way that the cross product defined by the following equation returns a normal vector oriented outwards the surface of the object at the position of vertex *V*:

$$N = (L_{N1} - L_{P1}) \times (L_{N2} - L_{P2})$$

According to the equation above Table 50 illustrates the correct identification of the -L_1- and -L_2- connectivity vectors attached to vertices located on one of the 6 discrete border faces (see Figure 50), given the imposed scanning orientation for a connectivity path *C*: i.e. *CCW* for external, respectively *CW* for internal surfaces. In order to satisfy the above equation the connectivity vectors of two consecutive vertices may change their ordering ($L_1 \rightarrow L_2$ and $L_2 \rightarrow L_1$). The exclamation marks from Table 50 indicate cases that are not possible.

The connectivity-wireframe has been coded using a new type of Freeman chain code extended to 3D, which is a typical coding method for the discrete space. For the MeshGrid representation, the discrete space is represented by the discrete (*u*,*v*,*w*) positions of the reference-grid to which the vertices are attached to.

The relative discrete border direction of two consecutive vertices along any scanning direction C_1 or C_2 (see Figure 49) may have only three different orientations, and therefore are coded on two bits only, having the meaning shown in Table 49(b).

Table 49 — Encoding of the relative border direction (orientation) of two consecutive vertices in (a) for the particular case when the vertices have the same discrete border direction, and in (b) the general case. The encoding of the discrete border direction of the starting vertex is given in (c)

Bits	Meaning
0	same discrete border orientation – case V1 and V2
1	broken connectivity (open meshes)

(a)

Bits	Meaning
00	same discrete border orientation – case V1 and V2
01	consecutive discrete border directions are rotated 90° CCW – case V3 and V4
10	consecutive discrete border directions are rotated 90° CW – case V5 and V6
11	broken connectivity (open meshes)

(b)

Bits	Meaning
000	reserved
001	Back face
010	Front face
011	Left face
100	Right face
101	Bottom face
111	Top face

(c)

A fourth value is added to indicate broken connectivities in open meshes.

Yet, in the particular case when the mesh is homogeneous quadrilateral, which is defined when variable *meshType*, initialized in the MeshGridDecoderConfig class as explained in subclause 4.2.3.2.2, yields the value QUADRI_MESH, and the *sameBorderOrientation* flag is set, then all the vertices are located on the same discrete border orientation with respect to the {*u*,*v*,*w*} direction of the reference-grid. In this case 1 bit suffices to encode the connectivity between two consecutive vertices, as illustrated in Table 49 (a).

Further, Table 50 illustrates the relation between the discrete position of two connected vertices and their discrete border direction.

Although there are four connectivity-vectors for each boundary vertex in the connectivity-wireframe, only the outgoing connectivity vectors (-L_{N1}- and -L_{N2}-) need to be present in the stream.

The decoding of the connectivity-wireframe requires that the starting vertex V_S of the connectivity-description (see subclause 4.2.3.3.1.1) be defined as an absolute reference to the grid, i.e. an absolute position (*u*,*v*,*w*) and discrete border direction. As shown in Table 49 (c), the discrete border direction requires 3 bits to be encoded. Each of the indices (*u*,*v*,*w*) defining the position of V_S can be encoded on *n*-bits, where *n* (n_u,n_v,n_w) depends on the largest ROI size which is $szDefROI_d + 1$, as explained in B.1.3:

$$n_d = \lfloor \log_2(szDefROI_d + 1) \rfloor + 1, \quad d \in \{u, v, w\}$$

4.2.3.3.1.1 Decoding the Connectivity bits

The decoding of a multi-resolution connectivity-wireframe consists basically in decoding sequentially each single-resolution connectivity-wireframe apart. A single-resolution connectivity-wireframe is encoded on ROI basis, be it a single ROI or several. The connectivity-wireframe from each ROI can be decoded in random order in view-dependent decoding scenarios. The coded connectivity-description from each ROI may consist of one or several patches. Each patch contains (1) a starting vertex identified by its discrete position (u,v,w) and discrete orientation (border direction), (2) followed by a sequence of relative connectivity symbols computed between an ancestor and its outgoing siblings. The traversal from the ancestor to the siblings is based on a first-in-first-out (FIFO) approach, as illustrated in Figure 51. The patch coding/decoding stops automatically when all the outgoing connectivities of the sibling vertices have been consumed inside the ROI. In order to avoid coding twice the connectivity of the vertices at the border in between two adjacent ROIs, the domain of the ROI is [*min*, *max*) for each of the {u,v,w} directions. For the last ROI in any direction $d \in \{u,v,w\}$, the domain of the ROI is [min, max] in direction d.

For each resolution level L the ROIs are coded in ascending order as given by the following formula:

$i + j \times nROIs_U^L + k \times (nROIs_U^L + nROIs_V^L)$, with $i \in [0, nROIs_U^L - 1]$, $j \in [0, nROIs_V^L - 1]$, $k \in [0, nROIs_W^L - 1]$. First, i is incremented, followed by j, and next by k.

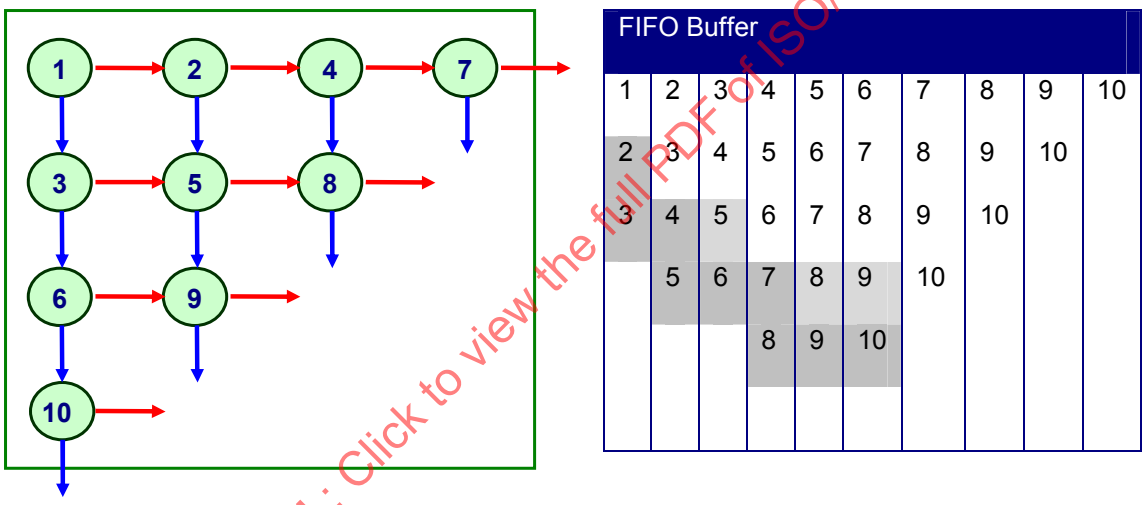


Figure 51 — Coding the LSD with a FIFO buffer

The decoding scenario for a patch illustrated by the example from Figure 51 is as follows: Retrieve the discrete position and discrete direction of the 1st vertex (starting vertex V_S) and store it in a *FIFO*. Consume the first vertex from the *FIFO*, retrieve the connectivity symbol leading to vertex 2 (reached via the outgoing link -L_{N1}-) and the connectivity symbol leading to vertex 3 (reached via outgoing link -L_{N2}-), compute the discrete positions of the vertices 2 and 3, and store both vertices 2 and 3 in the *FIFO*. Next pop vertex 2 from the *FIFO*, retrieve the two connectivity symbols leading to vertices 4 and 5, compute the discrete positions of the vertices 4 and 5, and insert them in the *FIFO*. In case the retrieved symbol leads to an outgoing connectivity vector pointing to a vertex V_O that was already visited before, or to a vertex V_O that lies outside the *ROI*, the visited vertex V_O will not be put in the *FIFO*. When consuming vertex 3 from the *FIFO*, retrieve the next two connectivity symbols leading to vertex 5 and 6, but insert in the *FIFO* only vertex 6, since vertex 5 has already been visited. The decoding scenario of a patch will stop when all vertices in the *FIFO* are consumed. Due to this implicit stopping criterion, only one byte-counter specifying the total length of all the coded patches belonging to the ROI is sufficient. For the other patches of the ROI, the same scenario is repeated.

4.2.3.3.1.2 Deriving the discrete positions of the vertices

In Table 50, the column header of the table has the labels of the border directions related to a vertex playing the role of an ancestor, while the row header displays the same border directions, but related to a sibling

vertex. The table gives all the possible combinations between the directions of the ancestor vertex and the sibling vertices. Combinations that are not possible have been shown with an exclamation mark. For any direction of the ancestor vertex, there are always only three valid border directions (with different combinations) for the sibling vertices.

For each valid case, the table gives also the position (*u,v,w*) in the reference-grid of the sibling vertex relative to the ancestor's vertex (only the position coordinate in the reference-grid of the sibling that differs from the ancestor's position is shown), and the connectivity vector from the sibling back to the ancestor.

Table 50 — Relation between the discrete border directions of two connected vertices and their discrete coordinates (u,v,w). Notice the change of indices of the connectivity vectors between adjacent vertices such that the normal vector points outwards

	Back		Front		Left		Right		Down		Up	
	-LN1 -	-LN2 -	-LN1 -	-LN2 -	-LN1 -	-LN2 -	-LN1 -	-LN2 -	-LN1 -	-LN2 -	-LN1 -	-LN2 -
Back	v -1 -LP1 -	w +1 -LP2 -	!	!	!	v -1 u +1 -LP1 -	same -LP1 -	!	same -LP2 -	!	!	w +1 u +1 -LP2 -
Front	!	!	w -1 -LP1 -	v +1 -LP2 -	!	same -LP2 -	v +1 u -1 -LP2 -	!	w -1 u -1 -LP1 -	!	!	same -LP1 -
Left	same -LP2 -	!	!	u +1 v +1 -LP2 -	w -1 -LP1 -	u +1 -LP2 -	!	!	!	w -1 v +1 -LP1 -	same -LP1 -	!
Right	u -1 v -1 -LP1 -	!	!	same -LP1 -	!	!	u -1 -LP1 -	w +1 -LP2 -	!	same -LP2 -	w +1 v -1 -LP2 -	!
Down	!	u -1 w +1 -LP1 -	same -LP1 -	!	same -LP2 -	!	!	v +1 w +1 -LP2 -	u -1 -LP1 -	v +1 -LP2 -	!	!
Up	!	same -LP2 -	u +1 w -1 -LP2 -	!	v -1 w -1 -LP1 -	!	!	same -LP1 -	!	!	v -1 -LP1 -	u +1 -LP2 -

4.2.3.3.1.3 The Cyclic Mode

The choice of the reference-grid as described in subclause 3.3.3.1.1 depends on the type of mesh that needs to be represented; it can range from uniform to highly irregular distributions of the grid points. In addition, the topology of the reference-grid can be defined as open, closed or folded. The closed reference-grid is designed by setting the grid points of the last reference-grid plane to the same coordinates as the corresponding grid points of the first reference-grid plane, in the "U" direction, or in two directions ("U" and "V") simultaneously. The folded reference-grid is can be cyclic in the "U" direction and connected in the "V" direction, i.e. half of the

grid points of the first (respectively last) reference-grid line are connected to the corresponding points of the other half by folding the line in the middle.

A connectivity-wireframe is cyclic when the connectivity vectors between the vertices span from the vertices located on the last (respectively first) reference-grid planes to those located on the first (respectively last) reference-grid planes. A connectivity-wireframe is folded when it is cyclic in the “U” direction and connected in the “V” direction, i.e. the connectivity-vectors from the vertices located on the first (respectively last) folded reference-grid line span to corresponding vertices on the folded line.

The cyclic/folded mode of a mesh is specified by the cyclic_folded variable from the MeshGridDecoderConfig class. As illustrated in Table 45, there is one non-cyclic mode, 4 cyclic/folded modes. A cyclic, respectively folded, mesh should be attached to a closed, respectively folded, reference-grid. When the mesh is cyclic, the connectivity-wireframe decoder will position each vertex that would normally be attached to the last reference-grid plane on the first reference-grid plane, in order to avoid the duplication of these vertices. In addition, when a mesh is folded, one vertex from each pair of duplicated vertices, i.e. pairs of vertices located on the folded reference-grid lines, is removed.

An example of a multi-resolution model cyclic in a single direction, in two directions and folded is shown in Figure 52, Figure 53 and Figure 54 respectively.

Note however, that even when the cyclic_folde mode flag is set to non-cyclic, it is still possible to decode a cyclic or folded connectivity-wireframe defined on top of either an open, closed or folded reference-grid, but in this situation the decoder will not take care of overlapping vertices.

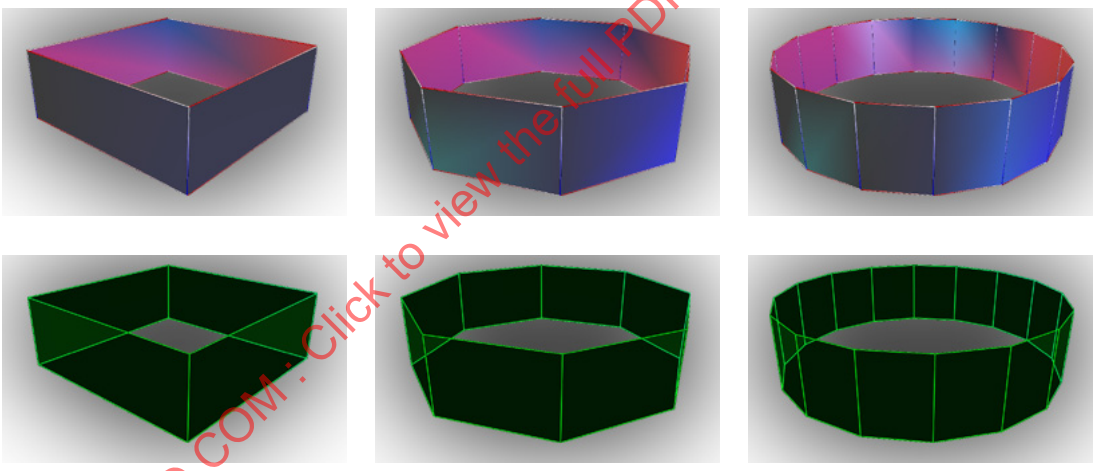


Figure 52 — Example of a multi-resolution mesh cyclic in one direction, and its corresponding reference-grid.

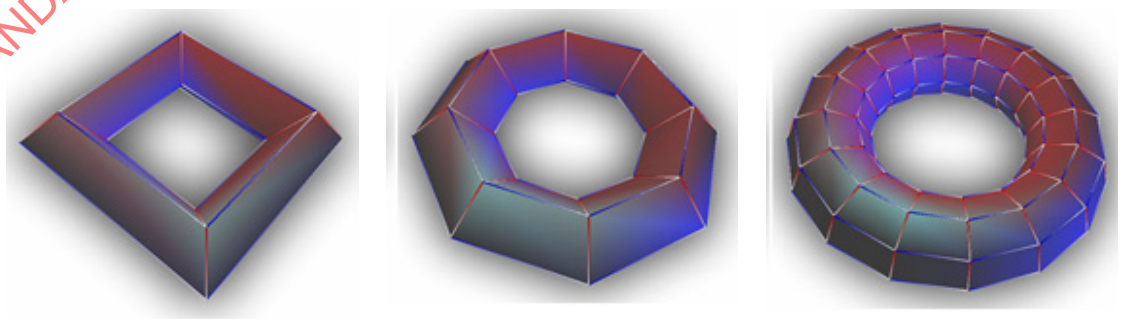
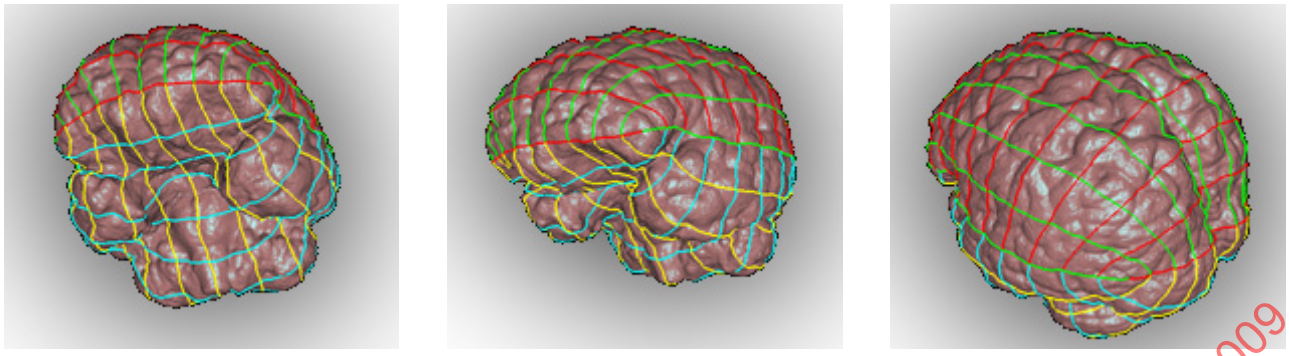


Figure 53 — Example of a multi-resolution mesh cyclic in two directions.



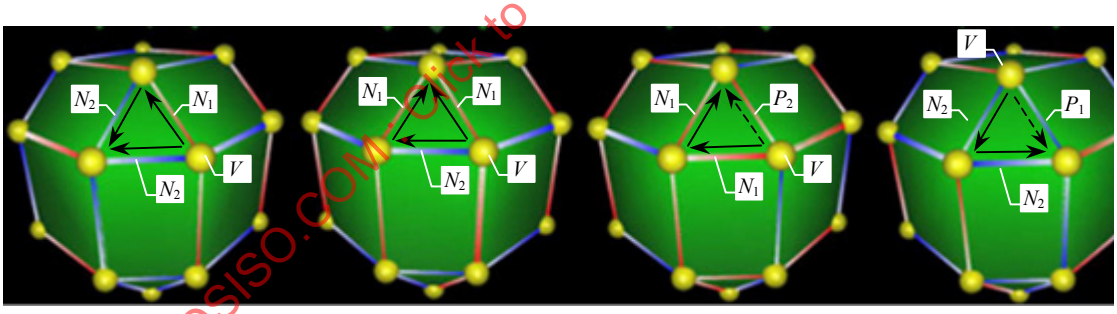
(a) (b) (c)

Figure 54 — Different views of a folded mesh.

4.2.3.3.1.4 Rules for deriving the surface primitives from the connectivity-wireframe

The polygonal representation of a MeshGrid model consists in triangulating the connectivity-wireframe, which can be seen as a union of connectivity circuits. A connectivity circuit corresponds to the shortest path formed by navigating from a starting vertex v_S to its neighbors following the connectivity vectors and back to vertex v_S.

In order to unambiguously identify the shape and the orientation of a connectivity circuit *CC*, such that the right surface primitive can be inserted, a set of connectivity rules should be designed. There are five types of primitives: triangles, quadrilaterals, pentagons, hexagons and heptagons, which can be identified by means of connectivity rules. As an example, a set of connectivity rules is given for the triangle primitive, Figure 55 showing the graphical cases, while Table 51 giving the connectivity conditions. The connectivity vectors drawn in Figure 55 use the same color convention as in Figure 50 and in Table 50. A complete set of rules for all five types of primitives is described in [68].



(a) (b) (c) (d)

Figure 55 — Images for the connectivity cases corresponding to the triangle primitive.

Table 51 — Connectivity cases for the triangle primitive

Case	Rule
(a)	N1N2= N2
(b)	N2 N1= N1
(c)	N1N1=P2 or N1N1N1=V
(d)	N2N2=P1 or N2N2N2=V

4.2.3.3.2 Reference-Grid Decoder

The general representation of the Reference-Grid description is a vector field ($x(u,v,w)$,$y(u,v,w)$,$z(u,v,w)$) defined on the regular discrete 3D space (u,v,w). Each component is coded separately, by means of a progressive multi-resolution algorithm based on a combination of a 3D wavelet transform and an intra-band wavelet coder, called Cube Splitting, which is the 3D extension of the SQP (Square Partitioning) algorithm described in [56]. This coding/decoding approach supports quality, resolution scalability, and ROI coding/decoding [2], [55], [56].

Subclause 4.2.3.3.2.1 describes the particular type of filters and down-sampling/up-sampling operations used for the wavelet decomposition and reconstruction, while the Cube Splitting algorithm is described in subclause 4.2.3.3.2.2.

4.2.3.3.2.1 The 3D Wavelet Decomposition

The same 3D-wavelet decomposition is applied independently to each of the $x(u,v,w)$, $y(u,v,w)$, $z(u,v,w)$ coordinates of the 3D reference grid.

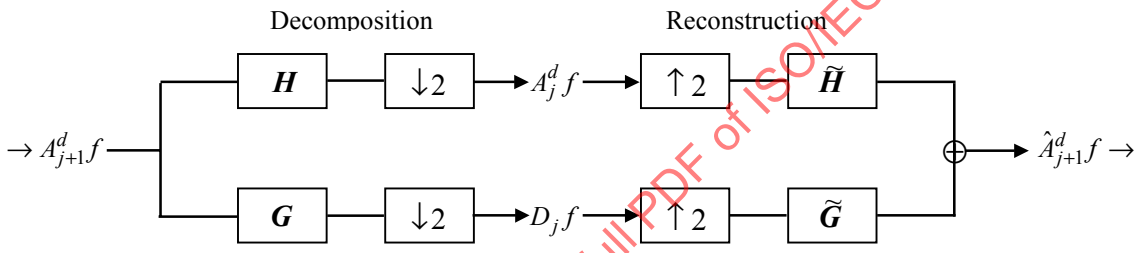


Figure 56 — Wavelet decomposition and reconstruction of a 1D signal.

For the 3D wavelet decomposition, the same analysis and synthesis 1D filters are used for each of the u, v, w directions. Two wavelet filters are supported, and the choice of the filter is specified by the *filterType* flag explained in 4.2.3.3.2. Conform to the block scheme [51] shown in Figure 56, the two analysis low-pass and band-pass wavelet filters are respectively:

$H(n)=\{1 \mid n=0\}$, $G(n)=\{-0.5,1,-0.5 \mid n=-1,0,1\}$, for filterType = 0, (eq1)

$H(n)=\{1 \mid n=0\}$, $G(n)=\left\{\frac{1}{16},0,-\frac{9}{16},1,-\frac{9}{16},0,\frac{1}{16} \mid n=-3,-2,-1,0,1,2,3\right\}$, for filterType = 1, (eq2)

The synthesis low-pass and band-pass wavelet filters are:

$\tilde{H}(n)=\{0.5,1,0.5 \mid n=-1,0,1\}$, $\tilde{G}(n)=\{1 \mid n=0\}$, for filterType = 0, (eq3)

$\tilde{H}(n)=\left\{-\frac{1}{16},0,\frac{9}{16},1,\frac{9}{16},0,-\frac{1}{16} \mid n=-3,-2,-1,0,1,2,3\right\}$, $\tilde{G}(n)=\{1 \mid n=0\}$, for filterType = 1. (eq4)

The wavelet filters are the same as the filters used for the hierarchical interpolation of the reference-grid points explained in subclause 3.3.3.3.1.

Further, in Figure 56, $A_j^d f(n)$ and $D_j f(n)$ are the discrete approximation and the detail signals respectively at the resolution $2^j, -L \le j \le -1$ of the input signal f, where L is the number of decomposition levels, and $A_0^d f(n)=f(n)$.

The low-pass filter H and the band-pass filter G are applied on the even and odd samples respectively of $A_{j+1}^d f(n)$ and the synthesis filters $\tilde{H}$ and $\tilde{G}$ are applied on the even and odd samples respectively of the low-pass and band-pass components $A_j^d f(n)$ and $D_j f(n)$.

The grid can be decoded at any resolution j if and only if the corners of the grid are stored in any $A_j^d f, -L \le j < 0$. There are some situations in which this constraint is not satisfied with the classical implementation of the pyramidal algorithm [51] discussed above. The wavelet analysis/synthesis filters given by (eq2) and (eq4) shall only be used when the constraint is satisfied for all decomposition levels $A_j^d f(n)$: the length of the discrete approximation signal $A_j^d f$ is odd. To solve this problem, a customized implementation of the pyramidal algorithm involves, in some situations, non-uniform down-sampling and up-sampling operations (see Figure 57(b,c)), coupled with analysis/synthesis filters that are different than the $H, G, \tilde{H}, \tilde{G}$ given by (eq1) and (eq3).

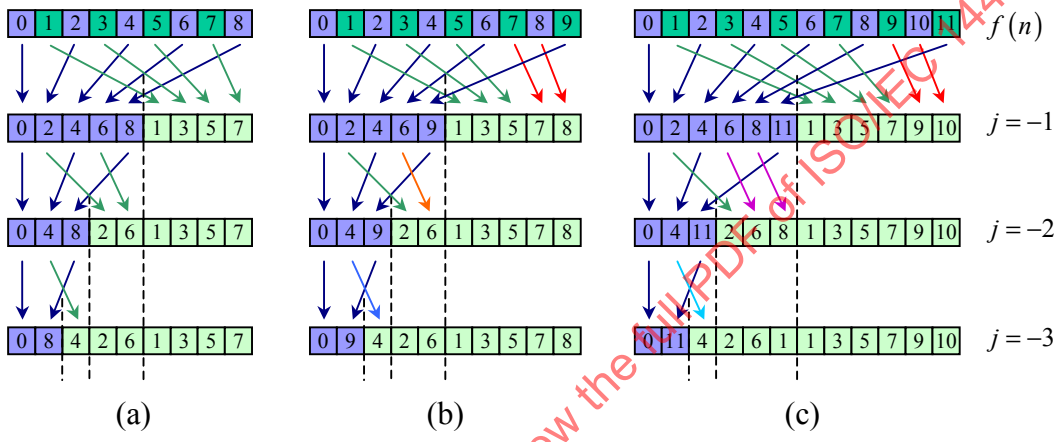


Figure 57 — Graphical illustration of the wavelet decomposition for odd and even-length signals.

As long as the length of the discrete approximation signal $A_j^d f$ is odd (see Figure 57(a)), there is no need to perform a non-uniform down-sampling operation, and the classical pyramidal algorithm can be used. If at some resolution level $r, -L < r \le 0$ the length of $A_r^d f$ is even, then apart from the common situation in which G given by (eq3) is used, the analysis band-pass filters $G^1_{|r|+1}, G^2_{|r|+1}$ have to be used as well, and non-uniform down-sampling has to be applied for the last samples. This operation has to be repeated for all the resolution levels $p, -L < p \le r$. The filter coefficients of $G^1_{|p|+1}, G^2_{|p|+1}$ depend on the length of the discrete approximation signal $A_p^d f$ (whether it is an odd or an even number) and on the resolution level p. The additional filters used to derive the detail $D_{p-1} f$ starting from the discrete approximation $A_p^d f$ are $G_p^1(n) = \{-c_2, 0, 1, -c_1 \mid n = -2, -1, 0, 1\}$, $G_p^2(n) = \{-c_4, 1, 0, -c_3 \mid n = -1, 0, 1, 2\}$ if the length of $A_p^d f$ is even, and $G_p^1(n) = \{-c_6, 1, -c_5 \mid n = -1, 0, 1\}$ if the length of $A_p^d f$ is odd. The constants $c_1, \ldots, c_6$ verify the relations $c_1 = 1 - c_2$, $c_3 = 1 - c_4$, $c_4 = 2c_2$, and $c_5 = 1 - c_6$. The constants c_2, c_6 are given by $c_2 = c_6 = x(n)/y(n)$, where $x(n)$ satisfies the recurrence $x(n) = x(n-1) + 2^{n-1}$, and $y(n)$ satisfies the recurrence $y(n) = y(n-1) + 2^{n-1}$ if the length of $A_p^d f$ is even, respectively $y(n) = y(n-1) + 2^n$ if the length of $A_p^d f$ is odd, with $x(0) = 1, y(0) = 3$ and $n = r - p$.

4.2.3.3.2.2 Intra-band wavelet coding algorithm

The coefficients generated by the wavelet transform undergo a scaling (see subclause 4.2.3.2.25) before they are coded with the Cube Splitting algorithm. The cube splitting algorithm is the straightforward extension of the

SQP coding algorithm [56] to 3D. The coding of the coefficients is done bit-plane by bit-plane in a depth-first strategy. With each bit-plane (b=0 is the lowest bit-plane), a threshold T_b is associated with $T_b=2^b$. Conceptually, there are two coding passes for each bit-plane (except for the first bit-plane): a significance pass and a refinement pass. A cube C is said to be significant for a certain threshold T if:

$$\underset{c(u,v,w)\in C}{MAX}\left(c(u,v,w)\geq T\right)$$

When a cube C is found non-significant, the NSG symbol is written into the bitstream. Otherwise, the SGN symbol is written and the cube is further split into 8 sub-cubes, visited in terms of the (u,v,w) reference axes in the following order: (0,0,0), (1,0,0), (0,1,0), (1,1,0), (0,0,1), (1,0,1), (0,1,1), (1,1,1).

During the splitting, each dimension is divided into two intervals. Let A be the smallest coordinate value, respectively B the highest coordinate value for a certain dimension {u,v,w} of the cube. The corresponding size S in that dimension is equal to $S=B-A+1$. The min values (A_{Left}, A_{Right}), max values (B_{Left}, B_{Right}), and the sizes (S_{Left}, S_{Right}) of the intervals can be computed as:

$$S_{Left}=\lfloor (S+1)/2 \rfloor, \quad S_{Right}=\lfloor S/2 \rfloor,$$
$$A_{Left}=A, \quad B_{Left}=A_{Left}+S_{Left}-1,$$
$$A_{Right}=A+S_{Left}, \quad B_{Right}=A_{Right}+S_{Rigth}-1.$$

If a cube has been found significant at a previous bit-plane, no SGN symbol is written in the stream. When the algorithm reaches the voxel level (i.e. no further splitting is possible), it first checks whether the voxel was already significant for higher bit-planes. If this is the case, the refinement symbol (BIT1 or BIT0) will be written into the stream. Otherwise, the algorithm will determine whether the voxel is significant for the current threshold value. If this is the case, the significant symbol SGN is written and is followed by the symbol for the sign (+,-) of the voxel-value. If both the previous cases are not valid, the non-significant symbol NSG will be written.

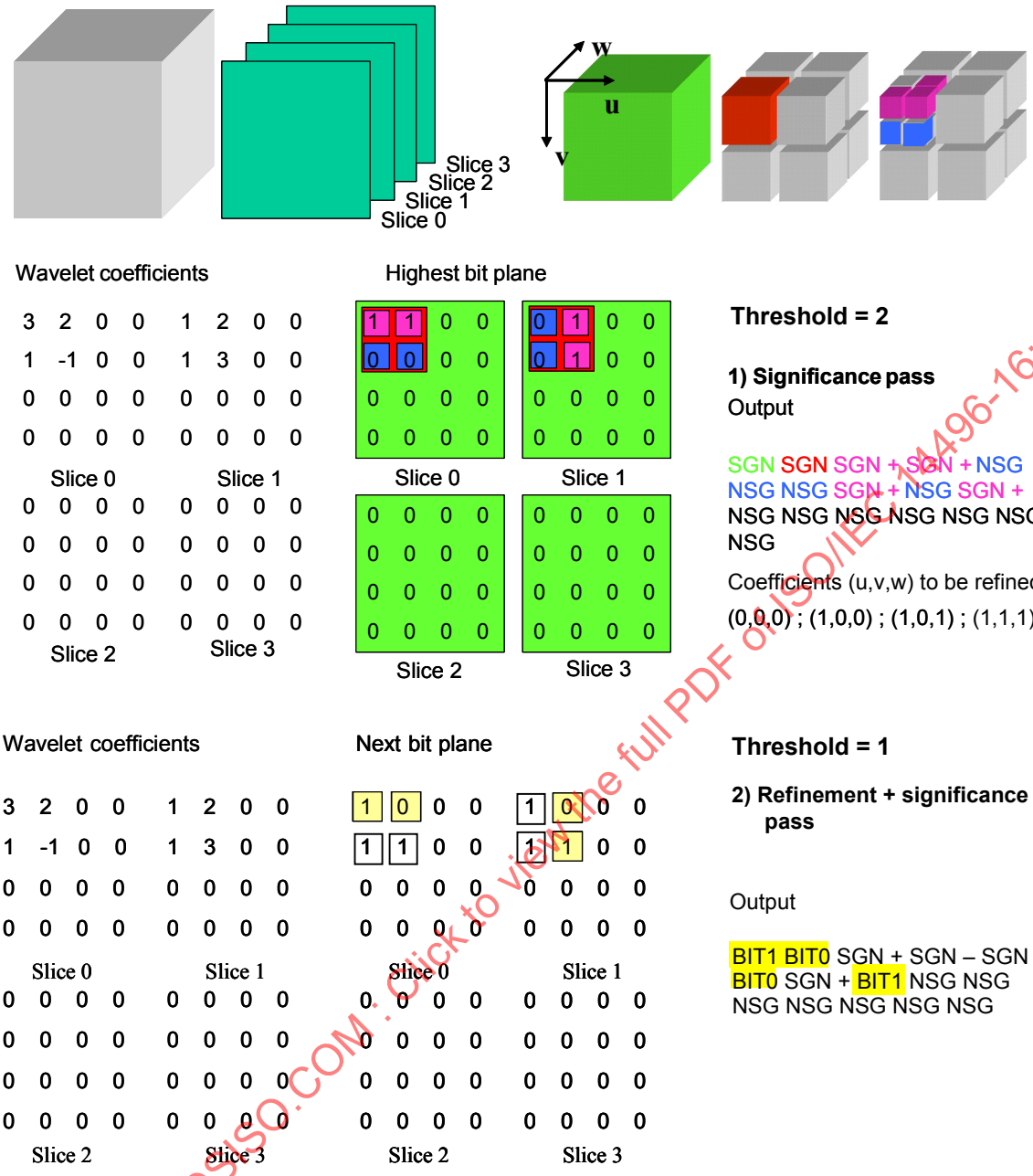


Figure 58 — Graphical representation of the significance and refinement coding passes.

The convention used in the MeshGrid stream for coding the symbols is shown in the following table:

Symbol	Bit value
SGN	1
NSG	0
BIT1	1
BIT0	0
+	1
-	0

4.2.3.3.3 Vertices-Refinement Decoder

The vertices are located at the intersection positions between the grid-lines and the object contour, at a certain ratio (the *offset*) in between two reference-grid points (see subclause 3.3.3.3.2). The *offset* (see Figure 23) has a default value of 0.5, but can be modified by the refinement description to fit a known position. The ratio can vary between [0,1).

The refinement description consists of two parts: (1) the inter-resolution refinement bit (*IRR*), and (2) the *offset*.

The significance of the *IRR* is shown in Figure 59. When decoding the next higher resolution level, some vertices may change their current discrete grid position (*u*,*v*,*w*), and migrate towards a neighboring grid position belonging to the higher resolution level. As a result of the migration, one of the {*u*,*v*,*w*} indices will change. The migration occurs along the grid line in the direction of the border. As illustrated in Figure 59, only the vertices with bit values larger than 0 will migrate.

For a specified resolution level (*l*) the default number of *IPR* bits for each vertex is one. For any vertex *v*(*u*,*v*,*w*), in the particular case that (1) the number of slices ($nSlices_d$) of any direction $d \in \{u,v,w\}$ of the immediate higher level (*l*+1) is a even number, (2) the grid position of vertex *v* in direction *d* is equal to ($nSlices_d - 2$), and (3) the discrete border direction of vertex *v* is oriented towards the positive axis of direction *d,* then there will be two *IRR* bits necessary to encode the repositioning value of vertex *v*, since two additional slices will be inserted between the slice at indices ($nSlices_d - 2$) and ($nSlices_d - 1$). This particular case has been explained in subclause 4.2.3.3.2.1, with a graphical example in Figure 57.

The *IRR* bits are necessary for all resolution levels, except for the last resolution level. A 2-bit flag (*repositionBits* explained in subclause 4.2.3.2.18) is present at each resolution level (except last) specifying whether default values are set for the *IRR* bits. If default values are specified, no *IRR* bits are encoded for that resolution level. Otherwise, the *IRR* bits are present in the stream. A default value of 0 or 1, indicates that all the *IRR* bits are 0 or 1, respectively.

For any ROI, the coding order in the stream of the *IRR* bits is the same as the order of the decoded vertices (see subclause 4.2.3.3.1.1) of the ROI.

If the number of quantization bits for the *offset* (*nBits* explained in subclause 4.2.3.2.23) is larger than 0, then the *offset* bits are present in the stream. Otherwise they are not available. If the full refine flag is set in the bitstream (*bFull* explained in subclause 4.2.3.2.23), then the *offset* bits are coded for each resolution level. Otherwise, the *offset* bits will only be coded for the last resolution level. The *offset* bits are stored bit-plane by bit-plane in the same order as the decoded vertices (see subclause 4.2.3.3.1.1).

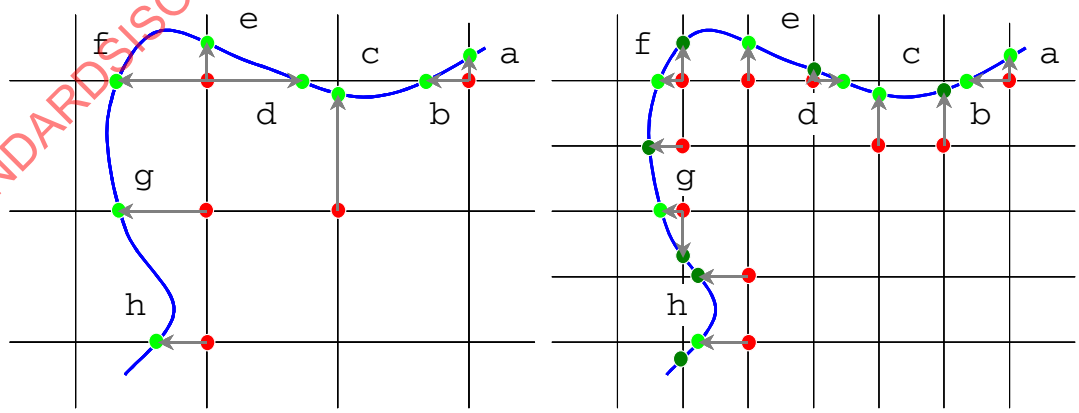


Figure 59 — Demonstration of vertex migration via the Inter-Resolution Refinement bits. The labelled vertices from the lower resolution Mesh-Description (left image) will migrate (some of them) to new reference-grid positions in the next higher resolution level (right image). For the sequence of vertices abcdefgh, the refinement bits are: 00110110, which means that vertices a, b, e, and h will not migrate while the others do.

4.2.3.3.4 ROI decoding for view-dependent functionality

As described in the beginning of subclause 4.2.1, the MeshGrid models are encoded at each resolution level in separate *region of interests* (ROIs), such that random decoding of the ROIs is possible to allow view-dependent scenarios.

The mesh description is encoded in the “spatial-domain” representation of the MeshGrid model, while the grid description is encoded in the “wavelet-domain” representation of the MeshGrid model. In a view-dependent scenario some parts of the model, which are identified in the “spatial-domain”, have to be decoded with a higher priority than the others. Therefore, a correspondence has to be determined between the “spatial-domain” ROIs (SROIs) and the “wavelet-domain” ROIs (WROIs) such that the appropriate grid description is decoded for the mesh description inside the SROIs. The correlation between a SROI defined at a certain resolution level and its WROIs is illustrated in Figure 60.

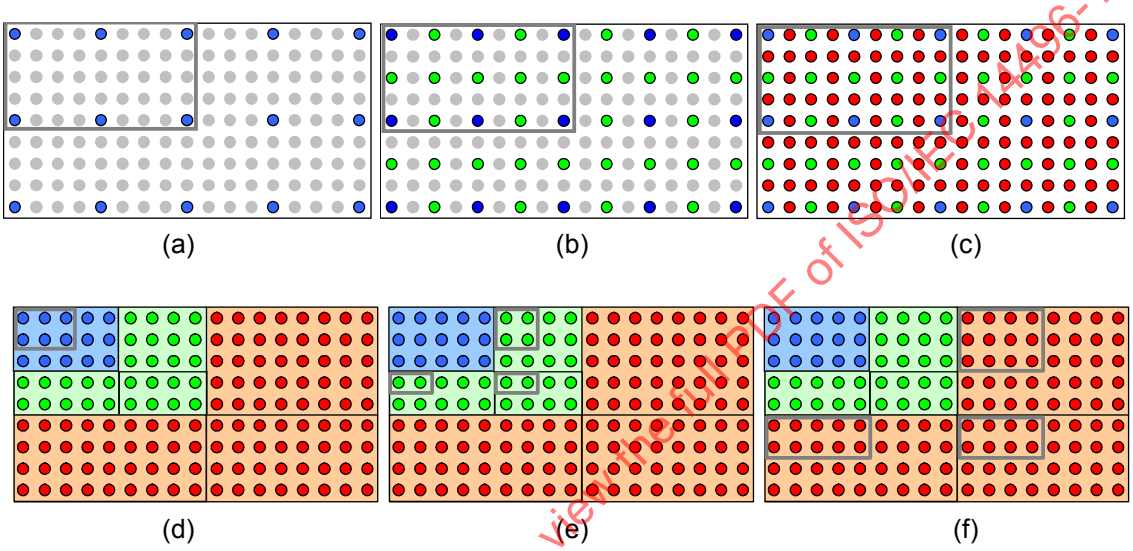


Figure 60 — A slice through the 3D reference-grid illustrating: (1) the distribution of the grid-points in the spatial domain (a, b, c) (resolution l, l+1, l+2) and in the wavelet domain (d, e, f), (2) the correspondence between a ROI defined at different resolutions in the spatial domain and its corresponding WROIs in the wavelet domain.

For each resolution level of the wavelet-decomposed reference-grid, the WROIs that are related to the same spatial domain SROI are grouped together and encoded as on tile. Therefore, for any resolution level *L* and direction $d \in \{U, V, W\}$ the number of tiles ($nTiles_d^L$) are equal to the number of ROIs ($nROIs_d^L$). Corresponding SROIs and tiles yield the same index $idx_d^L \in [0, nROIs_d^L - 1]$ when encoded in the bitstream. The tiles are coded with the Cube Splitting algorithm, described in subclause 4.2.3.3.2.2, in the same ascending order as the SROIs using the following formula:

$$i + j \times nROIs_U^L + k \times (nROIs_U^L \times nROIs_V^L),$$

with $i \in [0, nROIs_U^L - 1]$, $j \in [0, nROIs_V^L - 1]$, $k \in [0, nROIs_W^L - 1]$. First, i is incremented, followed by j, and next by k.

The number of ROIs is stored in the *nROIs* field of the MeshGridDecoderConfig class (see subclause 4.2.3.2.2) and the approach to compute the number of ROIs for the lower resolution levels and to uniformly distribute these ROIs is explained in subclause B.1.3.

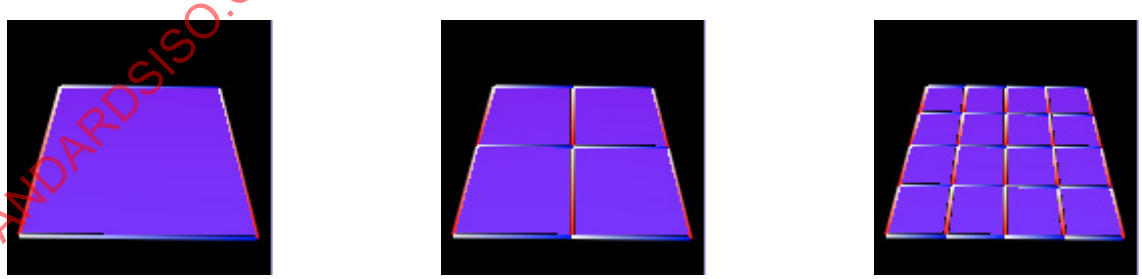
4.2.3.3.5 Decoding Quadrilateral MeshGrid

According to the *sameBorderOrientation* flag from the *MeshGridDecoderConfig* class (subclause 4.2.3.2.2) quadrilateral meshes can be classified into: (1) type1 (generic) – *sameBorderOrientation* equals '0' – when the vertices may have different discrete border directions (see Figure 50), and (2) type2 – *sameBorderOrientation* equals '1' – when all the vertices have the same discrete border direction.

For the type2 quadrilateral MeshGrid the *uniformSplit* flag from the *MeshGridDecoderConfig* class can have the value '1', which means that any higher resolution level quadrilateral mesh can be obtained from a lower resolution level mesh by uniformly splitting each quad recursively into four sub-quads as illustrated in Figure 61. In addition if the *hasConnectivityInfo* flag from the *MeshGridDecoderConfig* class has the value '0' – meaning that there is no connectivity-wireframe stored in the stream – then a default type2 quadrilateral connectivity-wireframe shall be constructed for the first resolution level. In this case the *uniformSplit* and *sameBorderOrientation* flags are set to '1'.

The reference-grid corresponding to the type2 quadrilateral MeshGrid with *hasConnectivityInfo* flag set '0' has the number of slices – *nSlices* from the *MeshGridDecoderConfig* class – in one of the directions {*u*,*v*,*w*} either equal to '2' – a double layer reference grid –, or equal to '1' – a single layer reference grid. For a double layer reference-grid the default type2 quadrilateral connectivity-wireframe shall be obtained by attaching a vertex to each reference-grid point belonging to the first layer of points i.e. the layer with the property that all the contained reference-grid points have one of the indices {*u*, *v*, *w*} equal to 0, and positioning these vertices on the reference-grid lines connecting the corresponding grid points from the first layer to the second layer. When the reference-grid is single layer, the default type2 quadrilateral connectivity-wireframe shall be obtained by attaching a vertex to each reference-grid point. The coordinates of the vertices can be computed as explained in subclause 3.3.3.3.2.

When the *hasRefineInfo* flag from the *MeshGridDecoderConfig* class (subclause 4.2.3.2.2) is set to '1' and there is a type2 quadrilateral MeshGrid with *hasConnectivityInfo* flag set '0', then the offsets (subclause 4.2.3.3.3) are specified only for the first resolution level of the mesh. The order of the offsets in the stream is the same as the indexing order of the reference-grid positions within the layer where the vertices are attached (the index corresponding to the layer is kept constant). The reference-grid points are indexed by incrementing two of the {*u*, *v*, *w*} indices in the order: *U* direction first, *V* direction second, and *W* direction third. The offset of each vertex corresponding to any higher resolution levels is computed as the average value of the offsets corresponding to the neighbour vertices' belonging to a lower resolution level. Namely, given V_1, V_2, V_3 and V_4 four vertices belonging to resolution level *l* (see Figure 61), the offset of a vertex *V* belonging to level *l* + 1 located on a grid line of level *l* is computed as the average value of the offsets of V_1 and V_2, respectively the offset of vertex *V*' belonging to level *l* + 1 located on a grid line of level *l* + 1 is computed as the average value of the offsets of V_1, V_2, V_3 and V_4.



(a) (b) (c)

Figure 61 — Example of a multi-resolution mesh with the connectivity-wireframe obtained by uniformly splitting each quad recursively into four sub-quads.

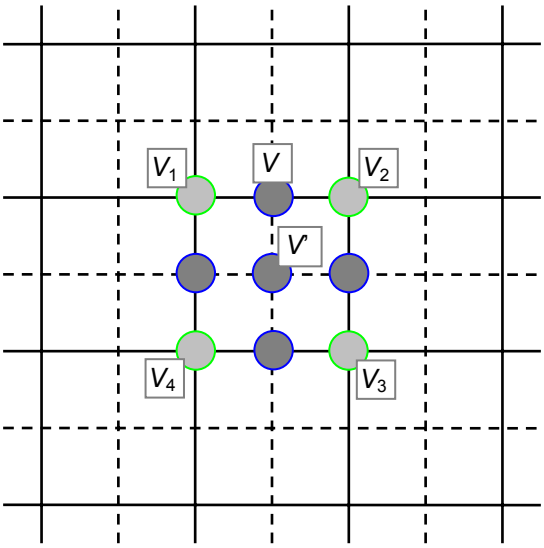


Figure 62 — Computation of the offsets for the type2 quadrilateral MeshGrid.

4.2.4 MultiResolution FootPrint-Based Representation

4.2.4.1 Downstream syntax

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

4.2.4.1.1 FootPrintSetDecoderSpecificInfo

4.2.4.1.1.1 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
   }
}
```

4.2.4.1.1.2 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 `log2(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.

4.2.4.1.2 FPBuildingDecoderSpecificInfo

4.2.4.1.2.1 Syntax

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

4.2.4.1.2.2 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.

4.2.4.1.3 FootPrintSet Message

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

4.2.4.1.3.2 Syntax

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

4.2.4.1.3.3 Semantics

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

4.2.4.1.4 FootPrint Message

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

4.2.4.1.4.1 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
         }
      }
   }
}
```

4.2.4.1.4.2 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 `log2(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.

4.2.4.1.5 FPNewVertices

4.2.4.1.5.1 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
    }
 }
}
```

4.2.4.1.5.2 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 specify 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*SignDelatX** by the **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.

4.2.4.1.6 IndexFootprintSet

4.2.4.1.7 Syntax

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

4.2.4.1.7.1 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.

4.2.4.1.8 FPBuildingParameters

4.2.4.1.8.1 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]
  }
}
}
```

4.2.4.1.8.2 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 may need to add a good detailed model that can be represent using a footprint-based representation. For example, one needs 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 the 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.

4.2.4.1.9 RoofBitStream

4.2.4.1.9.1 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
   }
}
```

4.2.4.1.9.2 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.

4.2.4.1.10 FacadeBitStream

4.2.4.1.10.1 Syntax

```
class FPFacade
{
   int(NbBitsFacadeWidth) WidthRatio
   int(2)  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
      }
   }
}
```

4.2.4.1.10.2 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.

4.2.4.2 Upstream syntax (for backchannel)

When specified as an upstream in corresponding ES descriptor, the MultiResolution Footprint Set stream has to be read according to the AFX Generic Backchannel syntax (see subclause 4.5.2).

4.3 Texture tools

4.3.1 Depth Image-Based Representation

4.3.1.1 Octree Compression

4.3.1.1.1 Overview

The OctreeImage node in Depth Image-Based Representation defines the octree structure and their projected textures. Each texture, stored in the images array, is defined through DepthImage node with SimpleTexture. The other fields of the OctreeImage node can be compressed by octree compression.

4.3.1.1.2 Octree

4.3.1.1.2.1 Syntax

```
class Octree ()
{
   OctreeHeader ();
   aligned bit (32)*  next;
   while (next == 0x000001C8)
   {
      aligned bit (32) octree_frame_start_code;
      OctreeFrame(octreeLevel);
      aligned bit (32)*  next;
   }
}
```

4.3.1.1.2.2 Semantics

The compressed stream of the octree contains an octree header and one or more octree frame, each preceded by **octree_frame_start_code**. The value of the **octree_frame_start_code** is always 0x000001C8. This value is detected by look-ahead parsing (next) of the stream.

4.3.1.1.3 OctreeHeader

4.3.1.1.3.1 Syntax

```
class OctreeHeader ()
{
   unsigned int (5) octreeResolutionBits;
   unsigned int (octreeResolutionBits) octreeResolution;
   int octreeLevel = ceil(log(octreeResolution)/log(2));

   unsigned int (3) imageNumBits;
   unsigned int (imageNumBits) numOfImages;
}
```

4.3.1.1.3.2 Semantics

This class reads the header information for the octree compression.

The **octreeResolution**, whose length is defined by **octreeResolutionBits**, contains the value of octreeResolution field of OctreeImage node. This value is used to derive the octree level.

The **numOfImages**, which is **imageNumBits** long, describes the number of images used in the OctreeImage node. This value is used for the arithmetic coding of image index for each voxel of the octree. If the value of **imageNumBits** is 0, then the image index symbols are not coded in OctreeFrame().

4.3.1.1.4 OctreeFrame

4.3.1.1.4.1 Syntax

```
class OctreeFrame (int octreeLevel)
{
   if(imageNumBits==0)  parentImage=255;
   else                 parentImage=0;
   for (int curLevel=0; curLevel < octreeLevel; curLevel++)
   {
      for (int voxelIndex=0; voxelIndex < nVoxelsInCurLevel; voxelIndex++)
      {
         int voxelSym = ArithmeticDecodeSymbol (contextID);
         if (parentImage == 0)
         {
            curImage = ArithmeticDecodeSymbol (imageContextID);
         }
      }
   }
   for (int voxelIndex=0; voxelIndex < nVoxelsInCurLevel; voxelIndex++)
      if (parentImage == 0)
         curImage = ArithmeticDecodeSymbol (imageContextID);
}
```

4.3.1.1.4.2 Semantics

This class reads a single frame of octree in a breadth first traversal order. Starting from the 1st node from the level 0, after reading every voxels in the current level, the number of voxels (nVoxelsInCurLevel) in the next level is known by counting all the 1's in each voxel symbol. In the next level, that number of voxels will be read from the stream.

For decoding of each voxel, an appropriate contextID is given, as described in subclause 4.3.1.1.6.

The parentImage and curImage are internal variables within each voxel in each level. The curImage is the image index for the current voxel in the current level. The parentImage is the image index of the parent, in the previous level, of the current voxel (the curImage of the parent voxel). When the bitstream does not contain the voxel image index information, the imageNumBits will be 0. In this case, the parentImage for every voxel shall be initialized to 255. Otherwise, the initial value of the parentImage for every voxel shall be 0.

If the image index for the parent voxel (parentImage) is not defined, then the image index for the current voxel (curImage) is also read from the stream, using the context for image index, defined by imageContextID. If a non-zero value is retrieved (the image index is defined), then this value will also be copied to every image index (curImage) of its children voxels in the following levels. In case of the root voxel, where curLevel is 0 and voxelIndex is 0, the value of parentImage depends on the value of imageNumBits in OctreeFrame(). If the imageNumBits is 0, then the parentImage of every voxel is a non-zero dummy value. This is the case where the voxel image indices are not used. If the imageNumBits has non-zero value, then the parentImage of every voxel is by default 0.

After decoding every voxel, the image index will be assigned to every leaf node of the octree that has no image index value assigned by the parent voxel yet. (zeros in leaf nodes can be interpreted by renderer in an arbitrary way).

The order of the reference images (streams) is essential: image indices for voxels assume the streams are numbered in a fixed order.

4.3.1.1.5 Adaptive Arithmetic Decoding

In this subclause, the adaptive arithmetic coder used in octree compression is described, using the C++ style syntactic description. aa_decode() is the function which decodes a symbol using a model specified through the array cumul_freq[] and is described in the Annex G of ISO/IEC 14496-1.

4.3.1.1.5.1 ArithmeticDecodeSymbol

This function prepares the appropriate cumul_freq[] to be used for aa_decode function, depending on the contextID. PCT is an array of probability context tables, as described in subclause 4.3.1.1.6.

```
int ArithmeticDecodeSymbol (int contextID)
{
   unsigned int MAXCUM = 1<<13;
   unsigned int ImageMAXCUM = 256;
   int *p, allsym, maxcum;

   if (contextID != imageContextID)
   {
      p = PCT[contextID];
      allsym = 256;
      maxcum = MAXCUM;
   }
   else
   {
      p = TexturePCT;
      allsym = numOfImages + 1;
      maxcum = ImageMAXCUM;
   }

   int cumul_freq[allsym];
   int cum=0;
   for (int i=allsym-1; i>=0; i--)
   {
      cum += p[i];
      cumul_freq[i] = cum;
   }
   if (cum > maxcum)
   {
      cum=0;
      for (int i=allsym-1; i>=0; i--)
      {
         PCT[contextID][i] = (PCT[contextID][i]+1)/2;
         cum += PCT[contextID][i];
         cumul_freq[i] = cum;
      }
   }
   return aa_decode(cumul_freq);
}
```

4.3.1.1.6 Decoding Process

The overall structure of decoding process is described in subclause 4.3.1.1.4. It shows how one obtains the TBVO voxels from the stream of bits that constitute the arithmetically encoded (compressed) TBVO model.

At each step of decoding process we must update the context number (i.e. the index of probability table), and the probability table itself. The probabilistic model is the union of all probability tables (integer arrays). j-th element of i-th probability table, divided by the sum of its elements, estimate the probability of occurrence of the j-th symbol in i-th context.

The process of updating the probability table is as follows. At the start, probability tables are initialized so that all the entries are equal to 1. Before decoding a symbol, the context number (ContextID) must be chosen. ContextID is determined from previously decoded data, as indicated in 4.3.1.1.6.1 and 4.3.1.1.6.2 below. When ContextID is obtained, the symbol is decoded using binary arithmetic decoder. After that, the probability table is updated, by adding adaptive step to the decoded symbol frequency. If the total (cumulative) sum of table elements becomes greater than the cumulative threshold, then the normalization is performed (see 4.3.1.1.5.1).

4.3.1.1.6.1 Context modeling of image symbol

Image symbol is modeled with only one context (i.e. only one probability table is used). The size of this table is equal to number numOfImages plus one. At the start, this table is initialized to all '1's. The maximum allowed entry value is set to 256. The adaptive step is set to 32. This combination of parameter values allows adapting to the highly variable stream of index numbers.

4.3.1.1.6.2 Context modeling of voxel symbol

There are 256 different voxel symbols, each symbol representing a 2×2×2 binary voxel array. 3D orthogonal transformation may be applied to these arrays, transforming the corresponding symbols into each other.

Consider a set of 48 fixed orthogonal transforms, that is, rotations by 90*n (n=0,1,2,3) degrees about the coordinate axis, and symmetries. Their matrices are given below, in the order of their numbers:

Orthogonal Transforms [48]=

{

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix},\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&0&1\\0&1&0\end{pmatrix},\begin{pmatrix}-1&0&0\\0&1&0\\0&0&1\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix},\begin{pmatrix}0&-1&0\\0&0&1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix},\begin{pmatrix}-1&0&0\\0&0&1\\0&1&0\end{pmatrix},$$

$$\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\1&0&0\\0&1&0\end{pmatrix},\begin{pmatrix}0&1&0\\-1&0&0\\0&0&1\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&-1&0\\-1&0&0\\0&0&1\end{pmatrix},\begin{pmatrix}0&0&1\\-1&0&0\\0&1&0\end{pmatrix},\begin{pmatrix}0&0&1\\0&1&0\\-1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\-1&0&0\\0&1&0\end{pmatrix},$$

$$\begin{pmatrix}0&-1&0\\0&0&1\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\0&1&0\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&1\\0&-1&0\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&1\\0&-1&0\\-1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\0&-1&0\\1&0&0\end{pmatrix},\begin{pmatrix}1&0&0\\0&-1&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&0&1\\0&-1&0\end{pmatrix},\begin{pmatrix}-1&0&0\\0&-1&0\\0&0&1\end{pmatrix},$$

$$\begin{pmatrix}1&0&0\\0&0&-1\\0&1&0\end{pmatrix},\begin{pmatrix}1&0&0\\0&1&0\\0&0&-1\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&-1\\1&0&0\end{pmatrix},\begin{pmatrix}0&1&0\\1&0&0\\0&0&-1\end{pmatrix},\begin{pmatrix}0&-1&0\\0&0&-1\\1&0&0\end{pmatrix},\begin{pmatrix}-1&0&0\\0&0&1\\0&-1&0\end{pmatrix},\begin{pmatrix}0&0&-1\\1&0&0\\0&-1&0\end{pmatrix},$$

$$\begin{pmatrix}0&0&1\\-1&0&0\\0&-1&0\end{pmatrix},\begin{pmatrix}0&-1&0\\0&0&1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\-1&0&0\\0&-1&0\end{pmatrix},\begin{pmatrix}0&-1&0\\0&0&1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&-1&0\\1&0&0\\0&0&-1\end{pmatrix},\begin{pmatrix}1&0&0\\0&0&-1\\0&-1&0\end{pmatrix},\begin{pmatrix}1&0&0\\0&-1&0\\0&0&-1\end{pmatrix},\begin{pmatrix}-1&0&0\\0&0&-1\\0&-1&0\end{pmatrix},$$

$$\begin{pmatrix}0&0&-1\\0&-1&0\\-1&0&0\end{pmatrix},\begin{pmatrix}0&0&-1\\0&1&0\\-1&0&0\end{pmatrix},\begin{pmatrix}-1&0&0\\0&-1&0\\0&0&-1\end{pmatrix},\begin{pmatrix}0&-1&0\\0&0&-1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&-1&0\\-1&0&0\\0&0&-1\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&-1\\-1&0&0\end{pmatrix},\begin{pmatrix}0&1&0\\-1&0&0\\0&0&-1\end{pmatrix},\begin{pmatrix}-1&0&0\\0&1&0\\0&0&-1\end{pmatrix},$$

}

There are 22 sets of symbols – called classes, - such that 2 symbols are connected by such a transform if and only if they belong to the same class. The coding method constructs PCT's (probability context tables) as follows: ContextID of a symbol equals either to the number of class to which its parent belongs, or to a combined number (parent class, current voxel position in the parent voxel). This allows to reduce the number of contexts greatly, and the time needed to gain meaningful statistics.

For each class, a single base symbol is determined (see Table 52), and for each symbol, the orthogonal transform that takes it into the base symbol of its class is precomputed (in actual encoding/decoding process, look-up table is used.). After the ContexID for a symbol is determined, the transform, inverse (i.e. transposed matrix) to the one taking its parent into the base element is applied.

Table 52 — An example of base symbol for each class

Class	Example of base symbol	Class order (Number of elements)
0	0	1
1	1	8
2	3	12
3	6	12
4	7	24
5	15	6
6	22	8
7	23	8
8	24	4
9	25	24
10	27	24
11	30	24
12	31	24
13	60	6
14	61	24
15	63	12
16	105	2
17	107	8
18	111	12
19	126	4
20	127	8
21	255	1

The context model depends on the number N of already decoded symbols:

For N < 512 there is only one context. The probability table is initialized to all '1'-s. The number of symbols in the probability table is 256. The adaptive step is 2. The maximum cumulative frequency is 8192.

For 512 ≤ N < 2560 (=2048+512), 1-context (in the sense that context number is single parameter, number of the class) model is used. This model uses 22 PCT's. ContextID is the number of the class to which the parent of the decoded node voxel belongs. This number can always be determined from the lookup table with base symbol and orthogonal transform, because the parent is decoded earlier than the child. Each of the 22 PCT's is initialized by the PCT from previous stage. The number of symbols in each probability table is 256. Adaptive step is 3. The maximum cumulative frequency is also 8192. After symbol is decoded, it is transformed using inverse orthogonal transform defined above.

When 2560 symbols are decoded, the decoder switches to 2-context (in the sense that the context number is now composed of the two parameters as explained below). This model uses 176 (=22*8, i.e. 22 classes by 8 positions) PCT's. ContextID here depends on the parent class and the position of the current node voxel in the parent nodevoxel. Initial probability tables for this model depend only on its context, but not position: for all 8 positions PCT is a clone of the PCT obtained for the given class at the previous stage. The number of symbols in each probability table is 256. Adaptive step is 4. The maximum cumulative frequency is also 8192. After symbol is decoded it is also transformed using the inverse orthogonal transform, as is in the previous model.

In the animated case, initial context model for every frame other than the first one is inherited from the previous frame.

4.3.2 PointTexture stream

4.3.2.1 Overview

The PointTexture compression is a tool to compress the PointTexture node efficiently. The decoder structure of the PointTexture compression is shown in Figure 63. The PointTexture decoder consists of header decoder and node decoder. The header information is decoded in the header decoder and is used in node decoder. The PointTexture decoder receives the arithmetic coded bitstream and restores the PointTexture node. The decoded PointTexture node specified in subclause 3.4.1.3 has the depth information and the color information.

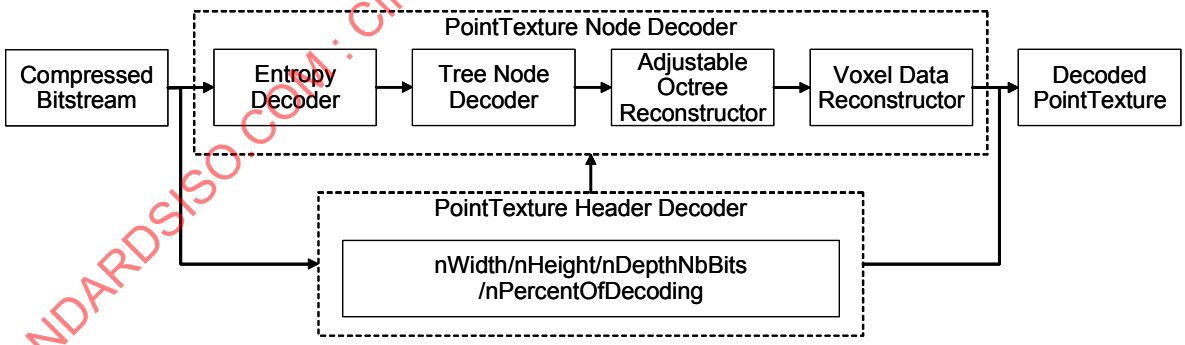


Figure 63 — Block diagram of decoder for PointTexture compression.

4.3.2.2 PointTexture class

4.3.2.2.1 Syntax

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

4.3.2.2.2 Semantics

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

4.3.2.3 PointTextureHeader class

4.3.2.3.1 Syntax

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

4.3.2.3.2 Semantics

nBitSizeOfWidth: This value indicates the bit size of nWidth.

nWidth: This value indicates the width of the PointTexture.

nBitSizeOfHeight: This value indicates the bit size of nHeight.

nHeight: This value indicates the height of the PointTexture.

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

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

4.3.2.4 PointTextureTreeNodes class

4.3.2.4.1 Syntax

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

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

   nIndex = popQ();
   nCount++;

   nSOP = decodeAAC(contextSOP);
   if(nSOP == 0)    // Split node decoding
   {
      nRegionRed = decodeAAC(contextRedOfRegion);
      nRegionGreen = decodeAAC(contextGreenOfRegion);
      nRegionBlue = decodeAAC(contextBlueOfRegion);
```

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

4.3.2.4.2 Semantics

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

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

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

pushQ: This function inserts a value into a queue.

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

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

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

popQ: This function extracts a value from the queue.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.2.5 Decoding Process

4.3.2.5.1 Overview

As shown in Figure 63, there are two parts to decode PointTexture. Those are header decoder and node decoder. The header decoder is to get the resolution information of PointTexture and the percent value how many tree nodes to decode. And the node decoding process is comprised of the following steps:

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

4.3.2.5.2 Header Decoding

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

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

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

Table 53 — Five labels for adjustable octree nodes

Labels	Comments
S	**Split:** The node is subdivided into 8 nodes
W	**White:** The node consists of all white voxels

B	**Fill black:** The node consists of, or is approximated by, all black voxels
P	**PPM:** The voxel values within the node are encoded by the PPM algorithm
E	**Empty:** The node has no voxel space

To explain the adjustable octree easily, the adjustable quad trees are adopted and used. Figure 64 and Figure 65 show the examples of the adjustable octrees. In the figures, the white voxels are represented by the white boxes and the white circles with W labels. And the non-white voxels are represented by the color boxes and the circles with B labels. If a node has children of S/W/B/P/E labels, the label of the node is S. If a node is to be decoded by PPM(Prediction by Partial Matching) decoding, the label is P. The empty nodes are represented by the dotted boxes and the dotted circles with E labels. Given a resolution, the full octree nodes and the empty nodes can be recognized and be found out. The decoder need not receive any bitstream or information for empty nodes, because it is possible to know the locations of all empty nodes with only the resolution information. In the adjustable octree, the parent node is subdivided into 8 children nodes regularly as equal as possible. Subdividing a parent node, the subdivision order of 8 children nodes is front left-top, front right-top, front left-bottom, front right-bottom, rear left-top, rear right-top, rear left-bottom, and rear right-bottom. In x, y, z axes, the length of each axis is divided into two sub-parts equally if possible. If it is not possible to divide equally, the length of one sub-part is one voxel longer than the length of the other sub-part. For example, Figure 64 shows that it is divided unequally in x-axis as 2 column voxels and 1 column voxel, but it is divided equally in y-axis as 2 row voxels and 2 row voxels.

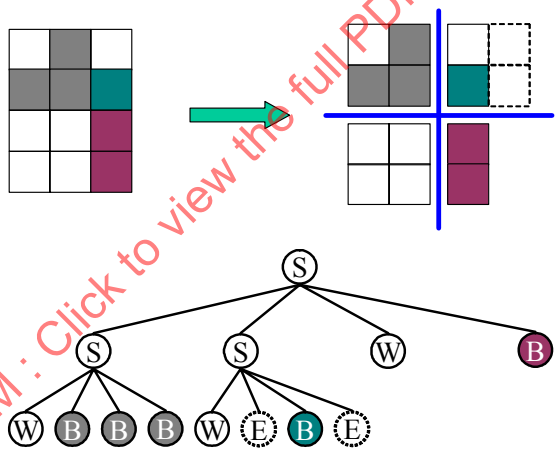


Figure 64 — Example of adjustable octree in the resolution of 3×4.

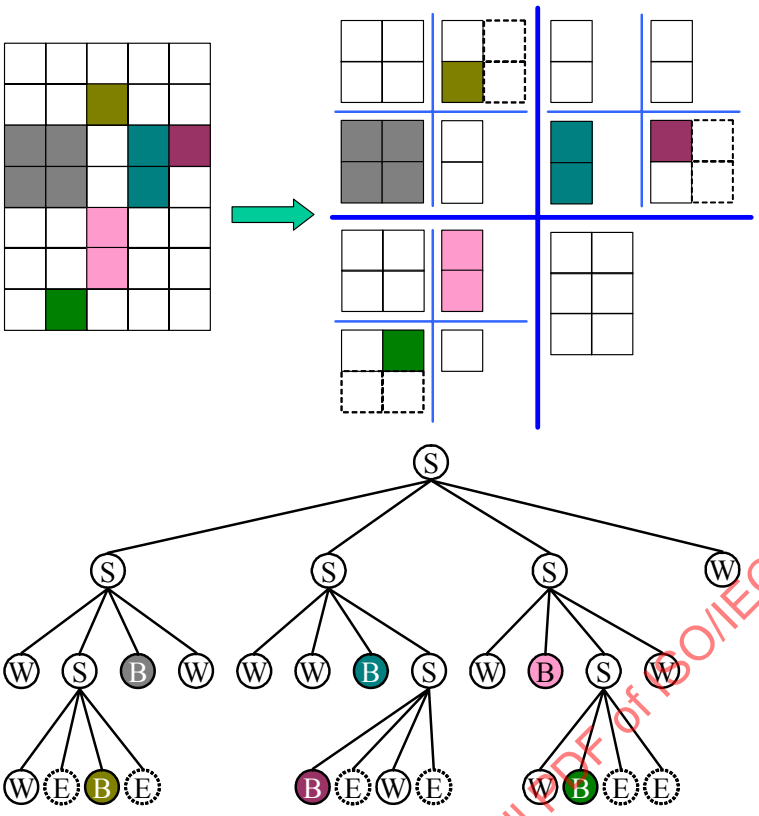


Figure 65 — Example of adjustable octree in the resolution of 5×7.

4.3.2.5.3 Entropy Decoding

The PointTexture decoder receives the bitstream and decodes the header information and the tree node information. To read the bitstream, the context-based adaptive arithmetic decoder is used as the entropy decoder [39]. The bitstream structure for compressed PointTexture is shown in Figure 66.

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

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

The header information contains the above resolution values and the percent of decoding. Each information of tree node is composed of two parts as SOP and DIB in Figure 66(b). SOP is one bit flag, which indicates that the tree node is split node or ppm node. If the node is split node, the bitstream structure is shown in Figure 66(c). Otherwise, the bitstream structure of ppm node is shown in Figure 66(d).

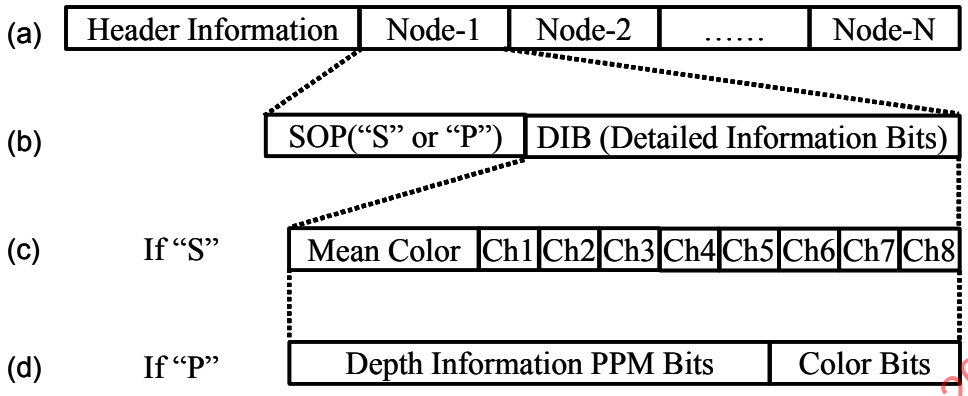


Figure 66 — Example of the bitstream structure for compressed PointTexture.

4.3.2.5.4 Tree Node Decoding

Given an octree resolution, the number of all tree nodes, N can be calculated and obtained. When the decoder received the bitstream of the tree nodes from root node to leaf nodes, it must know the decoding order in the tree nodes. A modified BFS (Breadth First Search) using a priority order queue can be used for the decoding order. If the decoder use a pure original breadth first search, then the progressive decoding is not possible, but the only sequential decoding is possible. So, modified BFS that is a modified BFS using a priority order queue is proposed and adopted to decode and show the progressive PointTexture. In the children's nodes, every first child node of the parent node is higher than the other child node of the parent nodes. Every second child node is higher than the node from the third to the eighth. Every eighth child node is most low than the other child node of the parent node in the priority. According to the children's priority from a parent node, the current decoding node can be notified to the decoder. Figure 67 shows an example of the decoding order for the tree nodes in Figure 64. In the figure, the empty E nodes are skipped and ignored in the decoding order. Figure 68 shows another example of the decoding order for the tree nodes in Figure 65.

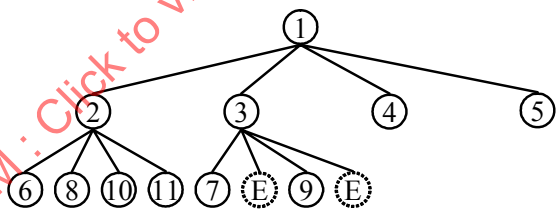


Figure 67 — Example of the decoding order for the tree nodes in Figure 65.

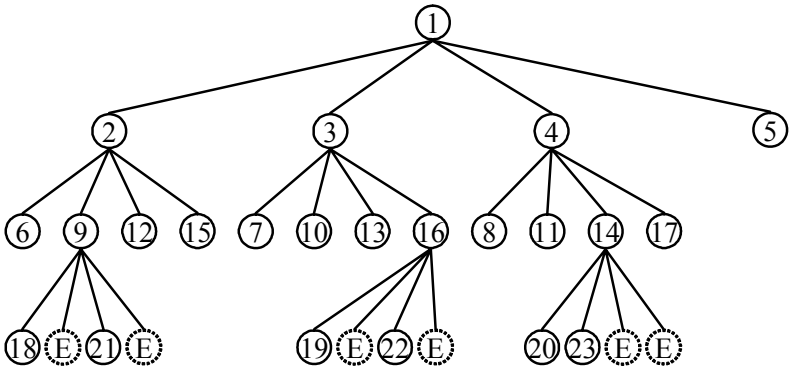


Figure 68 — Example of the decoding order for the tree nodes in Figure 66.

Figure 69 shows the block diagram of the tree node decoder. The decoding procedure is as follows. First, the decoder reads the SOP and finds out whether the tree node is split node or ppm node. Second, the decoder reads the DIB(Detailed Information Bits) and finds out where the voxels are and which colors are in each voxel. There are two cases, split node and ppm node. In case of split node, all color values of the children voxels are temporarily set to the average/mean color. These color values are updated when new sub colors of the children are decoded/received. In case of PPM node, the depth information of the voxel region is reconstructed using PPM decoding and the color information is also reconstructed using AAC(Adaptive Arithmetic Coder) decoding and the inverse-DPCM.

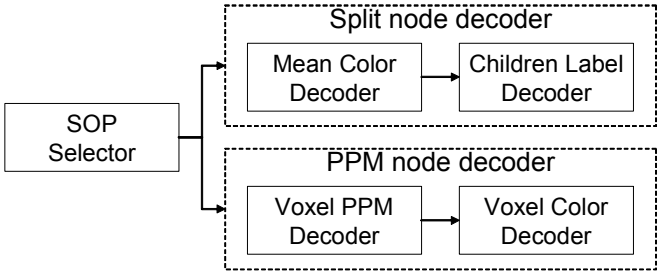


Figure 69 — Block diagram of tree node decoder.

In SOP Selector, the decoder reads the 1 bit flag, which indicates whether the node is the split node or PPM node. If the value is 0, it means split node. Otherwise, it means ppm node. In case of split node, the node is decoded as shown in Figure 66(c). Firstly, the average/mean R, G, B color values are decoded with the adaptive arithmetic decoder. Next, the children labels are also decoded with the adaptive arithmetic decoder. The child label can be black (1) or white (0). If the label is white, all children nodes of the node are white nodes. If the label is black, then the sub children nodes of the label are temporarily regarded as all black nodes. The sub children's nodes and colors can be known in detail when the next sub node of each label is decoded/reached.

In case of PPM node, the node is PPM decoded using the previous decoded voxel values as context. Figure 70 is an example of context. Voxels are represented as circles or squares. Gray circles means decoded black voxels and white circles means decoded white voxels. Question marked circles are not decoded voxels. A question marked black square shown in Figure 70 (b) is the voxel to be decoded. To decode the voxel as 0 or 1, ten neighboring voxels are used as context excluding three voxels which are marked with 'X'. Similar to the raster scan order, '0' is the previous decoded white circled voxel and '1' is the previous decoded gray circled voxel. The ten bits are used as the context of the squared voxels. In this example, the context of the squared voxels is '0111000011'. Using this context, the black squared voxel is decoded by the context-based adaptive arithmetic decoder.

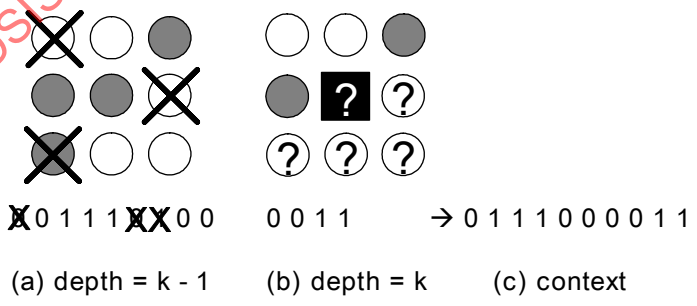


Figure 70 — Example of context: a voxel represented by a question marked black square in (b) is the voxel to be decoded and voxels represented by white circles and gray circles in (a) and (b) are used to make a context excluding three voxels which are marked with 'X'.

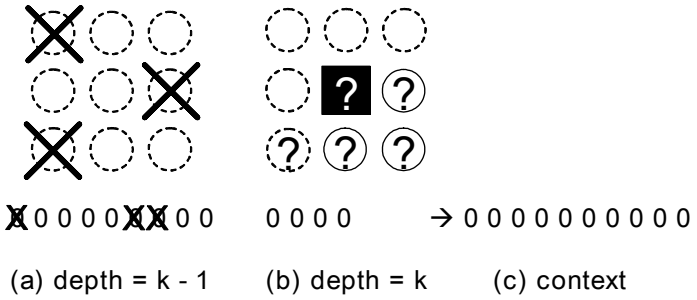


Figure 71 — Example of context: voxels represented by dotted circles in (a) and (b) are not existed voxels and a voxel to be decoded in (b) is located at a corner actually.

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

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

nRed = nDeltaRed + nRedPrevious , if nDeltaRed ≤ 255

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

nGreen = nDeltaGreen + nGreenPrevious , if nDeltaGreen ≤ 255

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

nBlue = nDeltaBlue + nBluePrevious , if nDeltaBlue ≤ 255

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

4.3.2.5.5 Adjustable Octree Reconstruction

In the tree node decoding, the labeled adjustable octree reconstruction is also performed. Figure 72 shows a simple example of the adjustable octree reconstruction in the resolution of 3×4. Figure 72 (b)(d)(f) show the reconstruction process in the decoder side. On the other hand, Figure 72 (a)(c)(e) shows the construction process in the encoder side. In Figure 72 (b)(d)(f), blue line box means the current decoding node and green line box means the decoding children nodes. In Figure 72 (b), the decoding node is split node and the decoding children nodes are B, B, W and B. In Figure 72 (d), the decoding node is PPM node and the decoding children nodes are W, B, B and B. In Figure 72 (f), the decoding node is split node and the decoding children nodes are W and B. In this case, E nodes are not decoded. They can only be recognized and found by the resolution information.

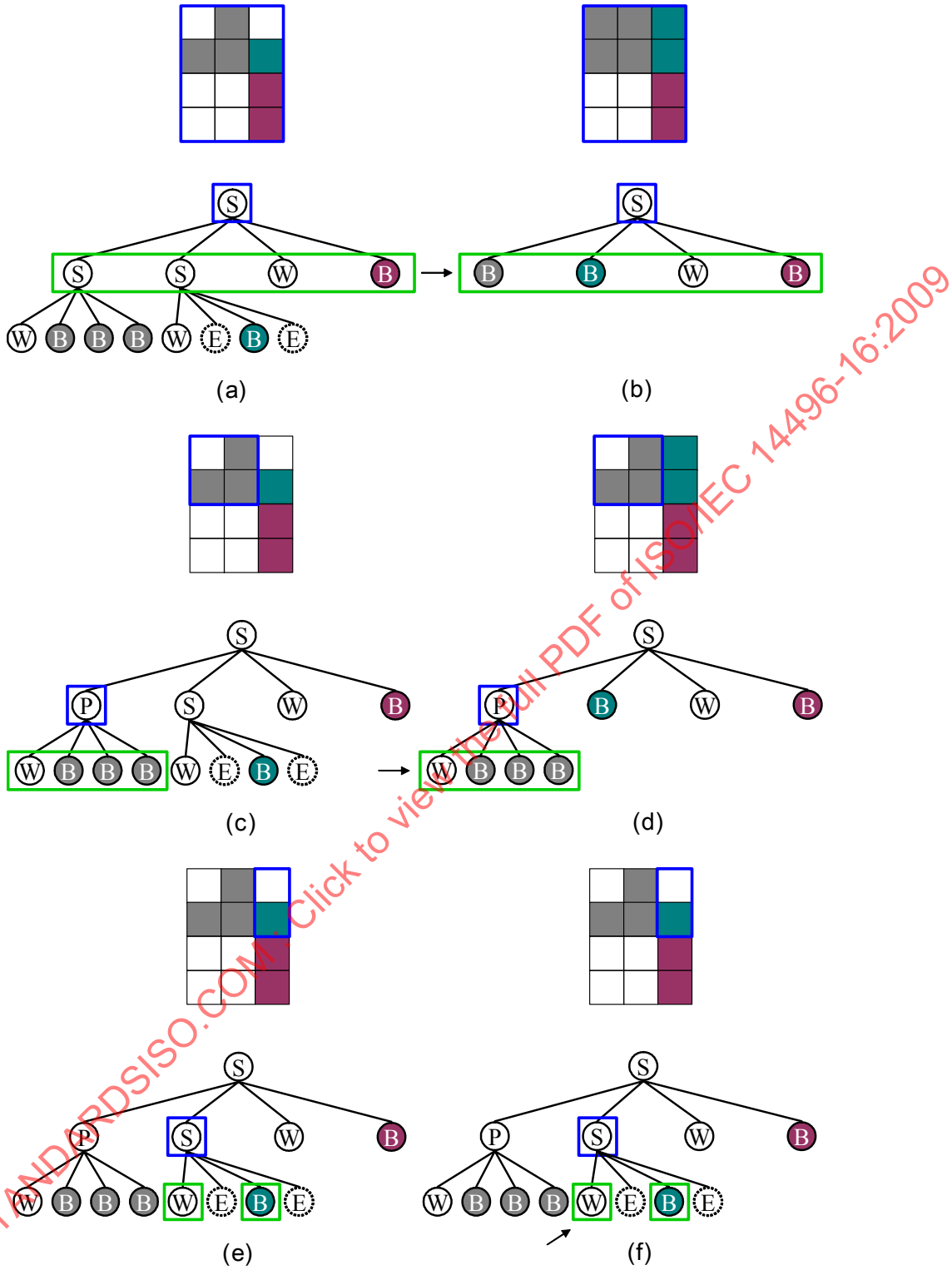


Figure 72 — Example of the adjustable octree construction/reconstruction.

4.3.2.5.6 Voxel Data Reconstruction

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

4.4 Animation tools

4.4.1 Bone-based animation

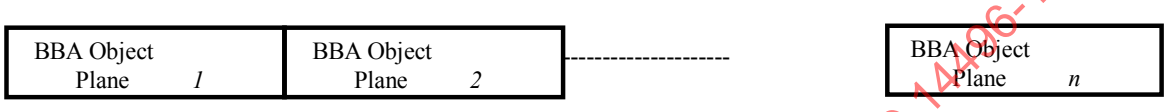
This tool compress a temporal sequence of bone(s) transforms.

4.4.1.1 Bone-Based Animation stream definition

4.4.1.1.1 Overview

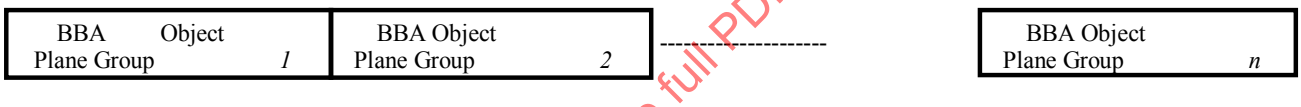
A BBA object is formed by a temporal sequence of BBA object planes as depicted below:

BBA object:

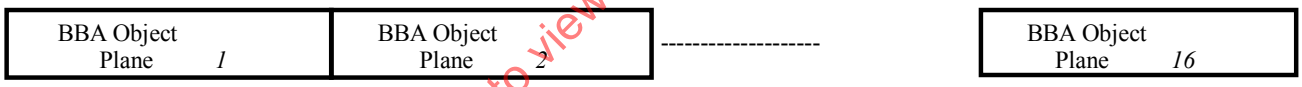


Alternatively, a BBA object can be formed by a temporal sequence of BBA object plane groups (called segments for simplicity), where each BBA object plane group itself is composed of a temporal sequence of 16 BBA object planes, as depicted in the following:

BBA object:



BBA object plane group:



When the alternative BBA object bitstream structure is employed, the bitstream is decoded by DCT-based BBA object decoding in the same manner as for the FBA stream. Otherwise, the bitstream is decoded by the frame-based BBA object decoding.

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

4.4.1.1.2 bba_object

4.4.1.1.2.1 Syntax

```
class bba_object {
  bit(23)* next;
  if (next==0)
    bit(32) bba_object_start_code;
  do{
    bba_object_plane bbaObPlane();
    bit(23)* next;
  }while (!((next==0)&&(next!=bba_object_plane_start_code)));
}
```

4.4.1.1.2.2 Semantics

In the `bba_object`, a start code is sent to enable resynchronization. The first 23 bits are read ahead, and stored as the integer `next`.

If `next` is 0 (in other words, the first 23 bits if the `bba_object_plane` are 0), the first 32 bits of the `bba_object` shall be read and interpreted as a start code that precedes the `bba_object`.

`bba_object_start_code` is equal to 000001EA in hexadecimal.

4.4.1.1.3 bba_object_plane

4.4.1.1.3.1 Syntax

```
class bba_object_plane() {
   bba_object_plane_header();
   bba_object_plane_data();
}
```

4.4.1.1.3.2 Semantics

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

4.4.1.1.4 bba_object_plane_header

4.4.1.1.4.1 Syntax

```
class bba_object_plane_header() {
  bit(23)* next;
  if (next==0)
    bit(32) bba_object_plane_start_code;
  bit(1) isIntra;
  if (isIntra) {
    bit(1) isFrameRate;
    if (isFrameRate)
      FrameRate rate;
    bit(1) isTimeCode;
    if (isTimeCode)
      unsigned int(18) timeCode;
  }
  bit(1) hasSkipFrames;
  if (hasSkipFrames)
    SkipFrames skip;
}
```

4.4.1.1.4.2 Semantics

In the `bba_object_plane_header`, a start code may be sent at each intra or predictive frame to enable resynchronization. The first 23 bits are read ahead, and stored as the integer `next`.

If `next` is 0 (in other words, the first 23 bits if the `bba_object_plane` are 0), the first 32 bits of the `bba_object_plane` shall be read and interpreted as a start code that precedes the `bba_object_plane`.

If the boolean `isIntra` is TRUE, the current animation frame contains intra-coded values, otherwise it is a predictive frame.

In intra mode, some additional timing information is also specified. The timing information obeys the syntax of the Facial Animation specification in ISO/IEC 14496-2. Finally, it is possible to skip a number of `bba_object_plane` by using the `FrameSkip` syntax specified in ISO/IEC 14496-2.

bba_object_plane_start_code = 000001EB in hexadecimal.

4.4.1.1.5 FrameRate

4.4.1.1.5.1 Syntax

```
class FrameRate {
   unsigned int(8) frameRate;
   unsigned int(4) seconds;
   bit(1) frequencyOffset;
}
```

4.4.1.1.5.2 Semantics

`frame_rate` is an 8-bit unsigned integer indicating the reference frame rate of the sequence.

`seconds` is a 4-bit unsigned integer indicating the fractional reference frame rate. The frame rate is computed as follows:

frame rate = (`frame_rate` + `seconds`/16).

`frequency_offset` is a 1-bit flag which when set to '1' indicates that the frame rate uses the NTSC frequency offset of 1000/1001. This bit would typically be set when `frame_rate` = 24, 30 or 60, in which case the resulting frame rate would be 23.97, 29.94 or 59.97 respectively. When set to '0' no frequency offset is present, i.e. if (`frequency_offset` ==1), frame rate = (1000/1001) * (`frame_rate` + `seconds`/16).

4.4.1.1.6 SkipFrame

4.4.1.1.6.1 Syntax

```
class SkipFrame {
   int nFrame = 0;
   do {
      bit(4) number_of_frames_to_skip;
      nFrame = number_of_frames_to_skip + nFrame;
   } while (number_of_frames_to_skip == 0b1111);
}
```

4.4.1.1.6.2 Semantics

`number_of_frames_to_skip` is a 4-bit unsigned integer indicating the number of frames skipped. If the `number_of_frames_to_skip` is equal to 15 (pattern "1111") then another 4-bit word follows allowing a skip of up to 29 frames (pattern "11111110") to be specified. If the 8-bits pattern equals "11111111", then another 4-bits word shall follow and so on, and the number of frames skipped is incremented by 30. Each 4-bit pattern of '1111' increments the total number of frames to skip with 15.

4.4.1.1.7 bba_object_plane_data

4.4.1.1.7.1 Syntax

```
class bba_object_plane_data() {
  bba_object_plane_mask bbaobplmsk();
  bba_object_plane_values bbaobplvls();
}
```

4.4.1.1.7.2 Semantics

bba_object_plane_data class contais information about the animation mask (class bba_object_plane_mask) and the animation values (class bba_object_plane_values).

4.4.1.1.8 bba_object_plane_mask

4.4.1.1.8.1 Syntax

```
class bba_object_plane_mask() {
  bit(5) NumberOfInterpolatedFrames; // NIF
  if (isIntra){
    bit(5) bba_quant;
    bit(3) pow2quant;
    bit(3) noOfControllerTypes;
    bit(10) NumberOfBones; //NSBB
    bit(10) NumberOfMuscles;//NSBM
    bit(10) NumberOfMorphs;//NMF
    for (bone=0;bone<NumberOfBones;bone++){
      bit(10) BoneIdentifier; //IDB
      bone_mask bnmask();
    }
    for (ms=0;ms<NumberOfMuscles;ms++){
      bit(10) MuscleIdentifier; //IDM
      bit(6) NumberControlPoints; //NCP
      bit(6) NumberKnots; //NK
      muscle_mask msmask();
    }
    for (mf=0;mf<NumberOfMorphs;mf++){
      bit(10) MorphIdentifier; //IDMF
      bit(6) NumberOfWeights; //NW
      morph_mask mfmask();
    }
  }
}
```

4.4.1.1.8.2 Semantics

NumberOfInterpolatedFrames **(NIF)** - 5-bit unsigned integer indicating the number of frames that have to be interpolated between the current frame and the received frame. If 0 (zero) the decoder pass the decoded frame to the animation engine.

bba_quant - a 5-bit unsigned integer used as the index to a **bba_scale** table for computing the quantisation step size of SBC values for predictive and DCT coding. If bba_object_coding_type is predictive, the value of **bba_scale** is specified in the following list:

bba_scale [0-31]= {0, 1, 2, 3, 5, 7, 9, 11, 14, 17, 20, 23, 27, 31, 35, 39, 43, 47, 52, 57, 62, 67, 72, 77, 82, 88, 94, 100, 106, 113, 120, 127};

If the bba_object_coding_type is DCT this is a 5-bit unsigned integer used as the index to a **bba_scale** table for computing the quantisation step size of DCT coefficients. The value of **bba_scale** is specified in the following list:

bba_scale [0 – 31] = { 1, 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 21, 25, 30, 35, 42, 50, 60, 72, 87, 105, 128, 156, 191, 234, 288, 355, 439, 543, 674, 836, 1039}

pow2quant – a 3-bit unsigned integer used as an exponent for computing the quantisation step size of SBC values.

The quantisation step is obtained as q=bba_scale[bba_quant]*2^pow2quant;

noOfControllerTypes – a 3 bits integer indicating the number of controller types in the BBA stream.

NumberOfBones (NSBB) - a 10-bit unsigned integer indicating the number of bones animated by the current frame.

NumberOfMuscles (NSBM) - a 10-bit unsigned integer indicating the number of muscles animated by the current frame.

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

BoneIdentifier (IDB) a 10-bit unsigned integer indicating a bone identifier; must correspond with a boneID field from a SBBone node from the scene graph.

MuscleIdentifier (IDM) a 10-bit unsigned integer indicating a muscle identifier; must correspond with a muscleID field from a SBMuscle node from the scene graph.

NumberControlPoints (NCP) A 6-bit unsigned integer indicating the number of control points of the muscle curve.

NumberKnots (NK) A 6-bit unsigned integer indicating the number of elements in the knot sequence of muscle curve.

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

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

4.4.1.1.9 bone_mask

4.4.1.1.9.1 Syntax

```
class bone_mask() {
  bit(1) IsTranslation_changed;
  bit(1) marker_bit;
  if (IsTranslation_changed){
    bit(1) IsTranslationOnX_changed;
    bit(1) IsTranslationOnY_changed;
    bit(1) IsTranslationOnZ_changed;
  }
  bit(1) IsRotation_changed;
  bit(1) marker_bit;
if (IsRotation_changed){
  bit(1) isQuaternion;
  if (!isQuaternion){
      bit(1) IsRotationOnAxis1_changed;
      bit(1) IsRotationOnAxis2_changed;
      bit(1) IsRotationOnAxis3_changed;
}else{
      bit(1) IsRotationOnQx_changed;
      bit(1) IsRotationOnQy_changed;
      bit(1) IsRotationOnQz_changed;
      bit(1) IsRotationOnQw_changed;
}
  }
  bit(1) IsScale_changed;
  bit(1) marker_bit;
  if (IsScale_changed){
    bit(1) IsScaleOnX_changed;
    bit(1) IsScaleOnY_changed;
    bit(1) IsScaleOnZ_changed;
  }
  bit(1) IsScaleOrientation_changed;
  bit(1) marker_bit;
  if (IsScaleOrientation_changed){
```

```
    bit(1) IsScaleOrientation AxisX_changed;
    bit(1) IsScaleOrientation AxisY_changed;
    bit(1) IsScaleOrientation AxisZ_changed;
    bit(1) IsScaleOrientation Value_changed;
  }
  bit(1) IsCenter_changed;
  bit(1) marker_bit;
  if (IsCenter_changed){
    bit(1) IsCenterOnX_changed;
    bit(1) IsCenterOnY_changed;
    bit(1) IsCenterOnZ_changed;
  }
}
```

4.4.1.1.9.2 Semantics

isTranslation_changed — 1 bit flag indicating, for the current bone, if at least one of the translation components is affected in the current frame

isTranslationOnX_changed — 1 bit flag indicating, for the current bone, if the translation on X axis is affected in the current frame

IsTranslationOnY_changed — 1 bit flag indicating, for the current bone, if the translation on Y axis is affected in the current frame

isTranslationOnZ_changed — 1 bit flag indicating, for the current bone, if the translation on Z axis is affected in the current frame

isRotation_changed — 1 bit flag indicating, for the current bone, if at least one of the rotation components is affected in the current frame

isRotationOnAxis1_changed — 1 bit flag indicating, for the current bone, if the rotation in respect with the first axis is affected in the current frame

isRotationOnAxis2_changed — 1 bit flag indicating, for the current bone, if the rotation in respect with the second axis is affected in the current frame

isRotationOnAxis3_changed — 1 bit flag indicating, for the current bone, if the rotation in respect with the third axis is affected in the current frame

isScale_changed — 1 bit flag indicating, for the current bone, if at least one of the scale components is affected in the current frame

isScaleOnX_changed — 1 bit flag indicating, for the current bone, if the scale on X axis is affected in the current frame

isScaleOnY_changed — 1 bit flag indicating, for the current bone, if the scale on Y axis is affected in the current frame

isScaleOnZ_changed — 1 bit flag indicating, for the current bone, if the scale on Z axis is affected in the current frame

isScaleOrientation_changed — 1 bit flag indicating, for the current bone, if at least one of the scaleOrientation components is affected in the current frame

isScaleOrientationAxisX_changed — 1 bit flag indicating, for the current bone, if the scaleOrientation on X axis is affected in the current frame

isScaleOrientationAxisY_changed — 1 bit flag indicating, for the current bone, if the scaleOrientation on Y axis is affected in the current frame

isScaleOrientationAxisZ_changed	1 bit flag indicating, for the current bone, if the scaleOrientation on Z axis is affected in the current frame
isScaleOrientationValue_changed	1 bit flag indicating, for the current bone, if the scaleOrientation angle value is affected in the current frame
isCenter_changed	1 bit flag indicating, for the current bone, if at least one of the center components is affected in the current frame
isCenterOnX_changed	1 bit flag indicating, for the current bone, if the center on X axis is affected in the current frame
isCenterOnY_changed	1 bit flag indicating, for the current bone, if the center on Y axis is affected in the current frame
isCenterOnZ_changed	1 bit flag indicating, for the current bone, if the center on Z axis is affected in the current frame

According to the received mask, the decoder sets-up the so-called “bone elementary mask” which contains 16 bits corresponding to: translation in X, translation in Y, translation in Z, rotation about first axis, rotation about second axis, rotation about third axis, scale factor along X, scale factor along Y, scale factor along Z, scaleOrientation axis X, scaleOrientation axis Y, scaleOrientation axis Z, scaleOrientation angle value, center deplacement in X, center deplacement in Y, center deplacement in Z. If one of the components is updated the elementary mask for it will be 1 (one); if not is 0 (zero).

If in the **SBBone** node the bone is defined as the end-effector of a kinematics chain, the translation component from the animation stream is used as the desired location of the end-effector. The rest of the bone transformation from the animation stream, if it exists, is ignored.

4.4.1.1.10 Muscle_mask

4.4.1.1.10.1 Syntax

```
class muscle_mask() {
  bit(1) IsControlPoints_Position_changed;
  bit(1) marker_bit;
  if (IsControlPoints_Position_changed){
    for (cp=1;cp<=NCP;cp++){
      bit(1) marker_bit;
      bit(1) IsControlPoint_onX_changed[cp];
      bit(1) IsControlPoint_onY_changed[cp];
      bit(1) IsControlPoint_onZ_changed[cp];
    }
  }
bit(1) IsControlPoints_Weight_changed;
  bit(1) marker_bit;
  if (IsControlPoints_Weight_changed){
    for (cp=1;cp<=NCP;cp++){
      bit(1) marker_bit;
      bit(1) IsControlPoint_weight[cp];
    }
  }
bit(1) IsKnot_changed;
  bit(1) marker_bit;
  if (IsKnot_changed){
    for (k=1;k<=NK;k++){
      bit(1) marker_bit;
      bit(1) IsKnot_changed[k];
    }
  }
}
```

4.4.1.1.10.2 Semantics

isControlPoints_Position_changed 1 bit flag indicating, for the current muscle, if at least one of the curve control points position is affected in the current frame

isControlPoints_onX_changed[cp] 1 bit flag indicating, for the current muscle, if control point index cp perform a translation on X axis in the current frame

isControlPoints_onY_changed[cp] 1 bit flag indicating, for the current muscle, if control point index cp perform a translation on Y axis in the current frame

isControlPoints_onZ_changed[cp] 1 bit flag indicating, for the current muscle, if control point index cp perform a translation on Z axis in the current frame

isControlPoints_Weight_changed 1 bit flag indicating, for the current muscle, if at least one of the curve control points weigh is affected in the current frame

isControlPoints_weight_changed[cp] 1 bit flag indicating, for the current muscle, if for the control point index cp a new weigh is transmitted in the current frame

isKnot_changed 1 bit flag indicating, for the current muscle, if at least one of the elements from the knot sequence is affected in the current frame

isKnot_changed[k] 1 bit flag indicating, for the current muscle, if for the knot element index k a new value is transmitted in the current frame

According to the received mask, the decoder set up the so-called “muscle elementary mask” which contains a variable number of bits corresponding to translation on X for the first control point of the curve, translation on Y for the first control point of the curve, translation on Z for the first control point of the curve, and so on for all the points, followed by the mask for control points weigh and the mask of knots. As for the bones, if a component is updated by the current frame the “muscle elementary mask” will be 1 (one); if not is 0 (zero).

4.4.1.1.11 morph_mask

4.4.1.1.11.1 Syntax

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

4.4.1.1.11.2 Semantics

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

4.4.1.1.12 bba_object_plane_values

4.4.1.1.12.1 Syntax

```
class bba_object_plane_values() {
  if (isIntra){
    bit(1) bba_object_coding_type;
    if (bba_object_coding_type==0){
      bit(1) bba_is_i_new_max;
      bit(1) bba_is_i_new_min;
      bit(1) bba_is_p_new_max;
      bit(1) bba_is_p_new_min;
      bba_new_minmax bbamnmx();
      bba_i_frame bbaifr();
    }else
      bba_i_segment bbaisg();
  }else{
    if (bba_object_coding_type==0)
      bba_p_frame bbapfr();
    else
      bba_p_segment bbapsg();
  }
}
```

4.4.1.1.12.2 Semantics

Bba_object_coding_type 1-bit integer indicating which coding method is used. Value 0 (zero) means the coding method is “predictive coding”, value 1 (one) means that the encoding method is DCT

Bba_is_i_new_max 1-bit flag which when set to ‘1’ indicates that a new set of maximum range values for I frame follows these 4, 1-bit fields

Bba_is_i_new_min 1-bit flag which when set to ‘1’ indicates that a new set of minimum range values for I frame follows these 4, 1-bit fields

Bba_is_p_new_max 1-bit flag which when set to ‘1’ indicates that a new set of maximum range values for P frame follows these 4, 1-bit fields

Bba_is_p_new_min 1-bit flag which when set to ‘1’ indicates that a new set of minimum range values for P frame follows these 4, 1-bit fields

If **Bba_is_i_new_max** is not specified the default value of 1860 is used.

If **Bba_is_i_new_min** is not specified the default value of -1860 is used.

If **Bba_is_p_new_max** is not specified the default value of 600 is used.

If **Bba_is_i_new_max** is not specified the default value of -600 is used.

4.4.1.1.13 bba_new_minmax

4.4.1.1.13.1 Syntax

```
class bba_new_minmax() {
  if (bba_is_i_new_max){
    for (sbc=1;sbc<=NUM_SBCs;sbc++){
      bit(1) marker_bit;
      if (sbc_mask[sbc])
        bit(5) bba_i_new_max[sbc];
    }
```

```
  if (bba_is_i_new_min){
    for (sbc=1;sbc<=NUM_SBCs;sbc++){
      bit(1) marker_bit;
      if (sbc_mask[sbc])
        bit(5) bba_i_new_min[sbc];
    }
  if (bba_is_p_new_max){
    for (sbc=1;sbc<=NUM_SBCs;sbc++){
      bit(1) marker_bit;
      if (sbc_mask[sbc])
        bit(5) bba_p_new_max[sbc];
    }
  if (bba_is_p_new_min){
    for (sbc=1;sbc<=NUM_SBCs;sbc++){
      bit(1) marker_bit;
      if (sbc_mask[sbc])
        bit(5) bba_p_new_min[sbc];
    }
}
```

4.4.1.1.13.2 Semantics

bba_i_new_max[one_sbc] 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the I frame for the current Skin&Bones component (SBC)

bba_i_new_min[one_sbc] 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the I frame for the current Skin&Bones component (SBC)

bba_p_new_max[one_sbc] 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the P frame for the current Skin&Bones component (SBC)

bba_p_new_max[one_sbc] 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the P frame for the current Skin&Bones component (SBC)

NUM_SBCs is the number of SBC in the current frame and is computed as:

$$NUM\_SBCs = \sum_{bn=1}^{NSBB} c_{bn} + \sum_{ms=1}^{NSBM} (3 * NCP_{ms} + NCP_{ms} + NK_{ms}) + \sum_{mf=1}^{NMF} NW_{mf}$$

with $c_{bn} = \begin{cases} 16, if \quad isQuaternion = 0 \\ 17, if \quad isQuaternion = 1 \end{cases}$ is the maximum number of the components for a bone transform.

sbc_mask contains NUM_SBCs elements and is obtained by concatenation of all the “bones elementary mask”, the “muscles elementary mask” and the "morphs elementary masks", affected in the current frame.

4.4.1.1.14 bba_i_frame

4.4.1.1.14.1 Syntax

```
class bba_i_frame() {
  for (sbc=1;sbc<=NUM_SBCs;sbc++){
    if (sbc_mask[sbc])
      aa_decode aad(isbc_Q[sbc], isbc_cum_freq[sbc]);
  }
}
```

4.4.1.1.14.2 Semantics

The SBC are quantized and coded by a predictive coding scheme. For each parameter to be coded in the current frame, the decoded value of this parameter in the previous frame is used as the prediction. Then the prediction error, i.e., the difference between the current parameter and its prediction, is computed and coded by arithmetic coding. This predictive coding scheme prevents the coding error from accumulating. The arithmetic decoding process is described in detail in Annex G from ISO/IEC 14496-1, *Coding of Audio-Visual Objects: Systems*.

The SBCs have the same precision requirement. Two information are used to obtain the quantisation step: the bba_quant which is an index parameter ranges from 0 to 31 and is an index to a sbc_pred_scale_table:

int sbc_pred_scale_table [32] = { 0, 1, 2, 3, 5, 7, 9, 11, 14, 17, 20, 23, 27, 31, 35, 39,

43, 47, 52, 57, 62, 67, 72, 77, 82, 88, 94, 100, 106, 113, 120, 127};

and the pow2quant which is en parameter ranges from 0 to 8.

The value of (SBC_QUANT = sbc_pred_scale_table [bba_quant]) == 0 has a special meaning, it is used to indicate lossless coding mode, so no dequantisation is applied. The quantisation stepsize is obtained as follows:

```
SBC_QUANT = sbc_pred_scale_table [bba_quant]
if (SBC_QUANT)
qstep = 2^pow2quant * SBC_QUANT
else
qstep = 1
```

The dequantized SBC'(t) is obtained from the decoded coefficient SBC''(t) as follows:

```
SBC'(t) = qstep * SBC''(t)
```

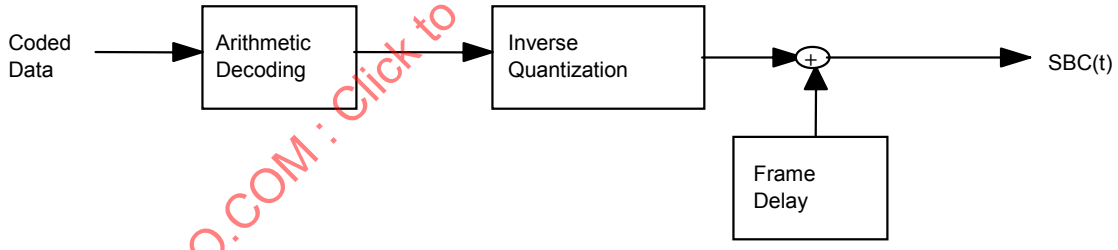


Figure 73 — SBC predictive coding.

4.4.1.1.15 bba_p_frame

4.4.1.1.15.1 Syntax

```
class bba_p_frame() {
  for (sbc=1;sbc<=NUM_SBCs;sbc++){
    if (sbc_mask[sbc])
      aa_decode aad(psbc_diff[sbc], psbc_cum_freq[sbc]);
  }
}
```

4.4.1.1.15.2 Semantics

See subclause 4.4.1.1.14.2.

4.4.1.1.16 bba_i_segment

4.4.1.1.16.1 Syntax

```
class bba_i_segment() {
  for (sbc=1;sbc<=NUM_SBCs;sbc++){
    if (sbc_mask[sbc]){
      decode_i_dc didc(dc_Q[sbc]);
      decode_ac   dac(ac_Q[sbc]);
    }
  }
}
```

4.4.1.1.16.2 Semantics

The bitstream is decoded into segments of SBCs, where each segment is composed of a temporal sequence of 16 SBCs object planes. The block diagram of the decoder is the following:

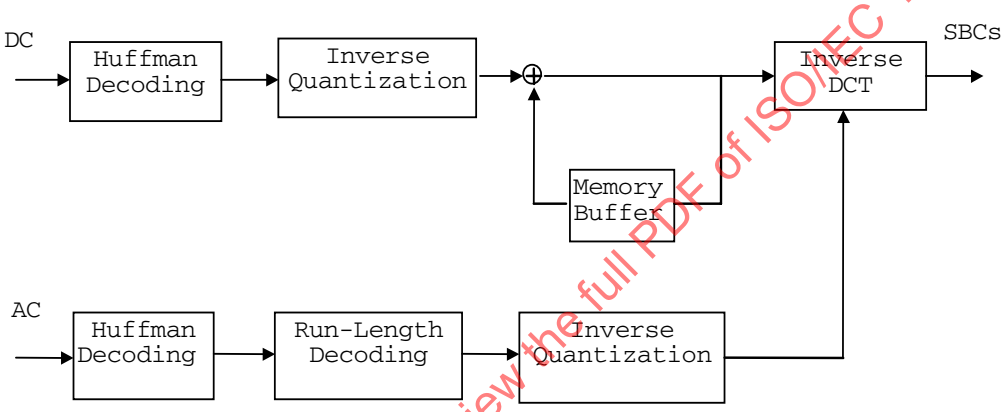


Figure 74 — Block diagram of the DCT-based SBCs decoding process.

The DCT-based decoding process consists of the following three basic steps:

- Differential decoding the DC coefficient of a segment.
- Decoding the AC coefficients of the segment.
- Determining the 16 SBCs values of the segment using inverse discrete cosine transform (IDCT).

A uniform quantisation step size is used for all AC coefficients. The quantisation step size for AC coefficients is obtained as follows:

```
qstep[i]  = sbc_scale[bba_quant] * 2^pow2quant;
```

with

```
static int sbc_scale [32] = { 0, 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 21, 25, 30, 35, 42,
50, 60, 72, 87, 105, 128, 156, 191, 234, 288, 355, 439, 543, 674, 836, 1023 };
```

The quantisation step size of the DC coefficient is one-third of the AC coefficients.

The DCT-based decoding process is applied to all SBCs. The DC coefficient of an intra coded segment is stored as a 16-bit signed integer if its value is within the 16-bit range. Otherwise, it is stored as a 31-bit signed integer. For an inter-coded segment, the DC coefficient of the previous segment is used as a prediction of the current DC coefficient. The prediction error is decoded using a Huffman table of 512 symbols. An "ESC" symbol, if obtained, indicates that the prediction error is out of the range [-255, 255]. In this case, the next

16 bits extracted from the bitstream are represented as a signed 16-bit integer for the prediction error. If the value of the integer is equal to -256*128, it means that the value of the prediction error is over the 16-bit range. Then the following 32 bits from the bitstream are extracted as a signed 32-bit integer, in twos complement format and the most significant bit first.

The AC coefficients, for both inter and intra coded segments, are decoded using Huffman tables. The run-length code indicates the number of leading zeros before each non-zero AC coefficient. The run-length ranges from 0 to 14 and proceeds the code for the AC coefficient. The symbol 15 in the run length table indicates the end of non-zero symbols in a segment. Therefore, the Huffman table of the run-length codes contains 16 symbols. The values of non-zero AC coefficients are decoded in a way similar to the decoding of DC prediction errors but with a different Huffman table.

4.4.1.1.17 bba_p_segment

4.4.1.1.17.1 Syntax

```
class bba_p_segment() {
  for (sbc=1;sbc<=NUM_SBCs;sbc++){
    if (sbc_mask[sbc]){
      decode_p_dc dpdc(dc_Q[sbc]);
      decode_ac   dac(ac_Q[sbc]);
    }
  }
}
```

4.4.1.1.17.2 Semantics

See subclause 4.4.1.1.16.

4.4.1.1.18 decode_i_dc

4.4.1.1.18.1 Syntax

```
class decode_i_dc(dc_q) {
  bit(16) dc_q;
  if (dc_q ==-256*128)
    bit(32) dc_q;
}
```

4.4.1.1.18.2 Semantics

See subclause 4.4.1.1.16.

4.4.1.1.19 decode_p_dc

4.4.1.1.19.1 Syntax

```
class decode_p_dc(dc_q_diff) {
  dc_q_diff; // decode Huffman
  dc_q_diff=dc_q_diff-256;
  if (dc_q_diff==-256)
    bit(16) dc_q_diff;
  if (dc_Q == 0-256*128)
    bit(32) dc_q_diff;
}
```

4.4.1.1.19.2 Semantics

See subclause 4.4.1.1.16.

4.4.1.1.20 decode_ac

4.4.1.1.20.1 Syntax

```
class decode_ac(ac_q[]) {
  this=0;
  next=0;
  while (next<15){
    bit(4) count_of_runs;
    if (count_of_runs == 15)
      next = 16;
    else {
      next=this+1+count_of_runs;
      for(n=this+1;n<next;n++)
        ac_q[i][n]=0;
      bit(8) ac_q[i][next];
      if (ac_q[i][next]==256)
        decode_i_dc didc(ac_q[i][next]);
      else
        ac_q[i][next]-=256;
      this=next;
    }
  }
}
```

4.4.1.1.20.2 Semantics

See subclause 4.4.1.1.16.

4.4.1.2 Bone-Based Animation stream encapsulation within BIFS-Anim

The Bone-based Animation (BBA) stream is connected to an **SBVCAnimation** node (see subclause 3.5.2.1.6) by using the BIFS-Anim mechanism (as for FBA) defined in ISO/IEC 14496-1.

The BIFS-Anim specific classes, ElemantaryMask and AnimationFrameData from subclauses 9.3.5.5 and 9.3.8.6 of ISO/IEC 14496-1, respectively, are updated as follows:

```
class ElementaryMask() {
  bit(BIFSConfiguration.nodeIDbits) nodeID;
  NodeUpdateField node = GetNodeFromID(nodeID);
  switch (node.nodeType) {
    case FaceType:
    break;
    case BodyType:
    break;
    case IndexedFaceSet2DType:
    break;
    case SBVCAnimationType:
    break;
    default:
    InitialFieldsMask initMask(node);
  }
}
class AnimationFrameData (AnimationMask mask) {
int i;
for (i=0; i<mask.numNodes; i++) {
  if (mask.isActive[i]) {
  NodeData node = mask.animNode[i]
  switch (node.nodeType) {
    case FaceType:
    FaceFrameData fdata; [ fdata ]
     break;
    case BodyType:
     BodyFrameData bdata; [bdata]
```

```
     break;
    case IndexedFaceSet2DType:
     Mesh2DframeData mdata;
    break;
    case SBVCAnimationType:
     bba_object_plane_data(); // see subclause 4.4.1.1.3
    break;
    default
    int j;
    for(j=0; j<node.numDYNfields; j++) {
      if (node.isAnimField[j])
AnimationField AField(node.field[node.dyn2all[j]],mask.isIntra);
      }
   }
  }
 }
}
```

4.4.2 Frame-based Animated Mesh Compression (FAMC) stream

4.4.2.1 Overview

FAMC is a tool to compress an animated mesh by encoding on a time basis the attributes (position, normals …) of vertices composing the mesh. FAMC is independent on the manner how animation is obtained (deformation or rigid motion). The data in a FAMC stream is structured in segments of several frames. Each segment can be decoded individually. Within a segment, a temporal prediction model, called *skinning*, is represented. The model is used for motion compensation inside the segment. The FAMC bitstream structure is illustrated in Figure 75.

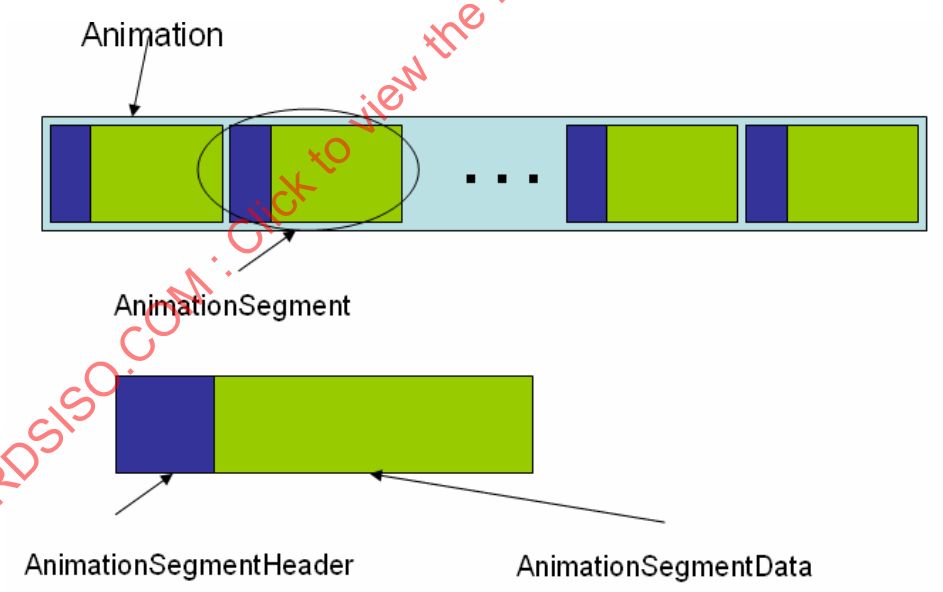


Figure 75 — FAMC bitstream structure.

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

An animation segment contains two types of information:

- A header buffer indicating general information about the animation segment (number of frames, attributes to be updated…).

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

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

Figure 75 illustrates the FAMC decoding process.

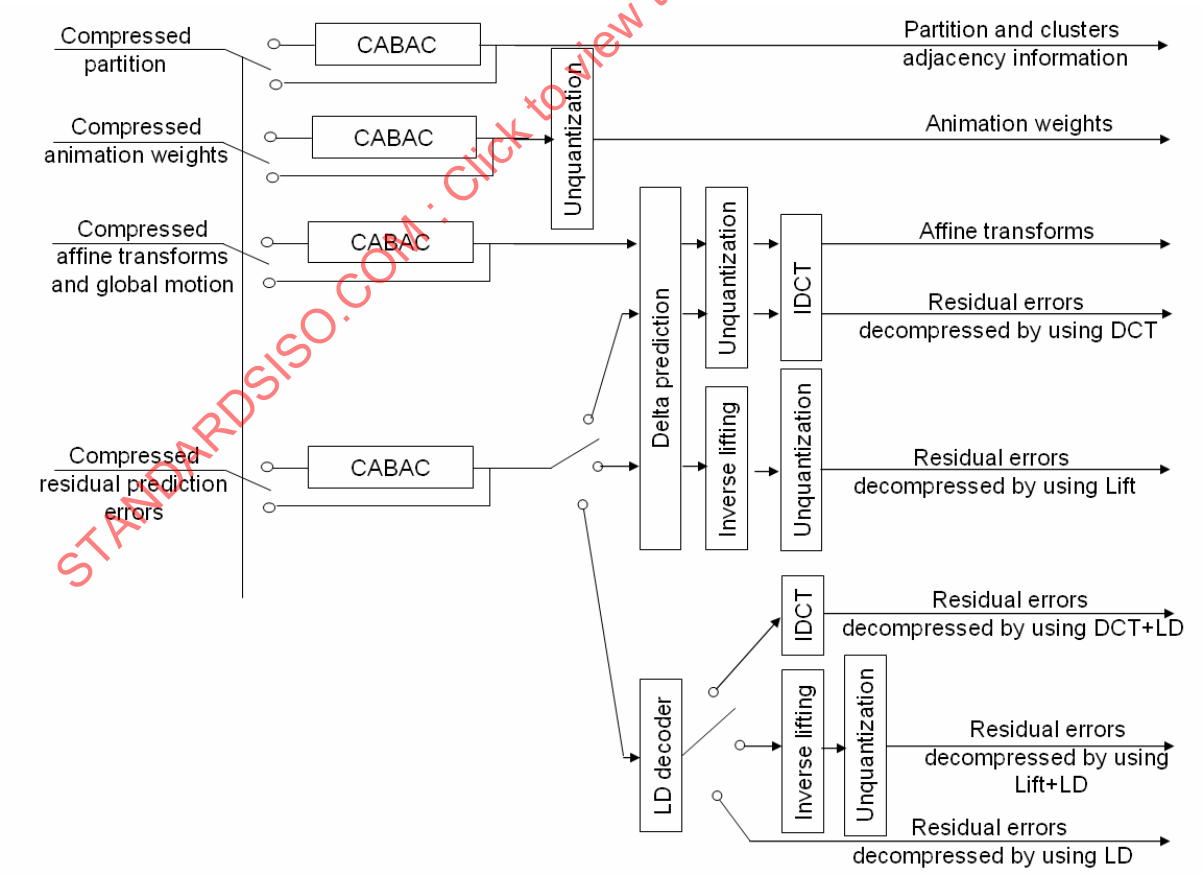


Figure 76 — FAMC decoding process.

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

4.4.2.2 FAMC inclusion in the scene graph

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

4.4.2.3 FAMC class

4.4.2.3.1 Syntax

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

4.4.2.3.2 Semantics

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

FAMCAnimationSegmentStartCode = 00 00 01 F0.

4.4.2.4 FAMCAnimationSegment class

4.4.2.4.1 Syntax

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

4.4.2.4.2 Semantics

FAMCAnimationSegmentHeader: contains the header buffer.

FAMCAnimationSegmentData: contains the data buffer.

4.4.2.5 FAMCAnimationSegmentHeader class

4.4.2.5.1 Syntax

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

4.4.2.5.2 Semantics

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

staticMeshDecodingType: a 8-bit unsigned integer indicating if the static mesh is encoded whithin the FAMC stream and which decoder should be used. The following table summarizes all possible configurations.

Table 54 — First frame decoding type: all possible configurations

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

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

animatedFields: a 4-bit mask indicating which fields are animated. The following table summarizes all possible configurations.

Table 55 — Animated fields: all possible configurations

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

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

Table 56 — Transform type: all possible configurations

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

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

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

normalsPredictionStrategy: a 2-bit mask indicating the prediction strategy for normals. The following table summarizes all possible configurations.

Table 57 — Normals prediction strategy: all possible configurations

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

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

colorsPredictionStrategy: a 2-bit mask indicating the prediction strategy for colors. The following table summarizes all possible configurations.

Table 58 — Color prediction strategy: all possible configurations

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

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

otherAttributesPredictionStrategy: a 4-bit mask indicating the prediction strategy for other attributes. The following table summarizes all possible configurations.

Table 59 — Other attributes prediction strategy: all possible configurations

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

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

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

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

4.4.2.6 FAMCAnimationSegmentData class

4.4.2.6.1 Syntax

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

4.4.2.6.2 Semantics

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

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

4.4.2.7 FAMCSkinningModel class

4.4.2.7.1 Syntax

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

4.4.2.7.2 Semantics

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

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

4.4.2.8 FAMCGlobalTranslationDecoder

4.4.2.8.1 Syntax

```
class FAMCGlobalTranslationDecoder {
        FAMCInfoTableDecoder globalTranslationCompensationInfo;
        FamcCabacVx3Decoder2 myGlobalTranslationCompensation(1, numberOfFrames);
}
```

4.4.2.8.2 Semantics

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

4.4.2.9 FAMCInfoTableDecoder class

4.4.2.9.1 Syntax

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

4.4.2.9.2 Semantics

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

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

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

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

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

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

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

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

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

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

4.4.2.10 FAMCCabacVx3Decoder2

4.4.2.10.1 Syntax

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

   EGk++;
   cutOff++;

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

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

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

   int absValues[V][F][3];
```

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

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

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

```
        }
        else{
           pred[v][c] = cabac.unary_exp_golomb_decode( cabac._dep, ccPred, 2 );
           if( pred[v][c] ){
              int predDimRes = cabac.unary_exp_golomb_decode( cabac._dep, ccPredDim, 2 );
              predDimRes <<= PRED_QUANT_BITS;
              if( predDimRes ){
                 const int largestAllowedPredDim = F + ( 1 << PRED_QUANT_BITS ) - 1;
                 if( previousDim[c] + predDimRes > largestAllowedPredDim ) {
                    predDimRes *= -1;
                 }
                 else if( previousDim[c] - predDimRes >= 0 ){
                    if( cabac.biari_decode_symbol_eq_prob( cabac._dep ) ){
                       predDimRes *= -1;
                    }
                 }
              }
              predDim[v][c] = predDimRes + previousDim[c];
              previousDim[c] = predDim[v][c];
           }
           else{
              predDim[v][c] = 0;
           }
        }
     }
  }
  // end the arithmetic coding engine
  cabac.biari_decode_final( cabac._dep );
}
```

4.4.2.10.2 Semantics

delta: reciprocal value of the quantization step size.

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

EGk: order of the Exp-Golomb binarization.

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

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

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

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

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

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

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

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

```
            values[v][d][c] += values[v-pred[v][c]][d][c];
          }
        }
      }
   }
```

4.4.2.11 FAMCAnimationPartitionDecoder

4.4.2.11.1 Syntax

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

4.4.2.11.2 Semantics

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

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

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

4.4.2.12 FAMCVertexInfoDecoder class

4.4.2.12.1 Syntax

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

4.4.2.12.2 Semantics

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

occurenceMinusOne: the number minus one of consecutive verticexIndex elements in partition.

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

4.4.2.13 FAMCAffineTransformsDecoder class

4.4.2.13.1 Syntax

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

4.4.2.13.2 Semantics

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

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

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

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

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

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

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

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

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

4.4.2.14 FAMCAnimationWeightsDecoder class

4.4.2.14.1 Syntax

```
class FAMCAnimationWeightsDecoder {
   unsigned int(8) numberOfQuantizationBits;
```

```
   float(32) minWeights;
   float(32) maxWeights;
   unsigned int(32) compressedWeightsBufferSize;

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

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

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

4.4.2.14.2 Semantics

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

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

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

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

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

numberOfBits = (int) (log((double) numberOfClusters-1)/log(2.0)+ 1.0);

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

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

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

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

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

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

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

The decoding process is composed of three steps:

- Vertices selection decoding. First, the CABAC context is initialized with value 61. Then, the one dimensional array filter of size numberOfVertices is decoded by using the CABAC function biari_decode_symbol.
- Clusters adjacency decoding. The CABAC context is initialized with value 61. For each cluster k, the number of its neighbours is decoded by using the CABAC function unary_exp_golomb_decode. Then, the index of each neighbour is decoded by using the CABAC function biari_decode_symbol_eq_prob. Each index is represented by its binary representation on numberOfBits bits.
- Weights decoding. The CABAC context is initialized with value 2. The weights are decoded bit-plane per bit-plane using the CABAC function biari_decode_symbol. In order to retrive the values of weights, the quantization process needs to be reversed.

4.4.2.15 FAMCAllResidualErrors

4.4.2.15.1 Syntax

```
class FAMCAllResidualErrors {
switch (transformType) {
  case 0 : // Lifting
  case 1 : // DCT
    if (animatedFields & 1)
FAMCInfoTableDecoder coordResidualErrorsInfo;
    if (animatedFields & 2)
FAMCInfoTableDecoder normalResidualErrorsInfo;
    if (animatedFields & 4)
FAMCInfoTableDecoder colorResidualErrorsInfo;
    if (animatedFields & 8)
FAMCInfoTableDecoder otherAttributesResidualErrorsInfo;
    do{
      if (animatedFields & 1)
FAMCCabacVx3Decoder2 coordErrorsLayerLift(numberOfVertices, numberOfCoefficientsPerLayer);
      if (animatedFields & 2)
FAMCCabacVx3Decoder2 normalErrorsLayerLift(numberOfNormals, numberOfCoefficientsPerLayer);
      if (animatedFields & 4)
FAMCCabacVx3Decoder2 colorErrorsLayerLift(numberOfColors, numberOfCoefficientsPerLayer);
      if (animatedFields & 8)
FAMCCabacVx3Decoder2 otherAttributesErrorsLayerLift(numberOfOtherAttributes,
numberOfCoefficientsPerLayer);
     bit(32)* next;
     }
     while (next==FAMCAnimationSegmentStartCode);
break;
        case 2: // LD
          FAMCLDDecoder allErrorsLD(numberOfVertices, numberOfFrames, animatedFields);
          break;
        case 3: // Lift + LD
          FAMCLDDecoder allErrorsLiftLD(numberOfVertices, numberOfFrames,animatedFields);
```

```
            break;
          case 4: // DCT + LD
            FAMCLDDecoder allErrorsDCTLD(numberOfVertices, numberOfFrames, animatedFields);
            break;
    }
}
```

4.4.2.15.2 Semantics

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

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

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

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

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

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

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

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

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

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

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

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

To recover the original values the decoder applies:

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

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

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

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

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

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

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

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

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

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

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

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

where

$$N_t^v = \frac{U_t^v \times W_t^v}{\left\| U_t^v \times W_t^v \right\|} + n_t^v ,$$ $$W_t^v = \frac{\sum_{k=1}^{numberOfClusters} \omega_v^k A_t^k W^v}{\left\| \sum_{k=1}^{K} \omega_v^k A_t^k W^v \right\|}$$, (U^v, W^v, N_1^v) is the orthonormal basis of IR^3, N_1^v is the normal of vertex v at the first frame and n_t^v is the normal residual errors of vertex v at frame t.

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

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

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

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

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

The other attributes decoding is identical to normal decoding.

4.4.2.16 FAMCLDDecoder class

4.4.2.16.1 Syntax

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

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

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

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

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

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

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

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

      unsigned int(32) compressedFrameSizeLD;
      for (int layerNumber=0; layerNumber<numberOfEncodedLayers; ++layerNumber){
      if (animatedFields & 1)
         FAMCCabacVx3Decoder
            resCoords(numberOfVerticesInLayer[layerNumber], hasCoordsPredBits);
      if (animatedFields & 2)
         FAMCCabacVx3Decoder
            resNormals(numberOfVerticesInLayer[layerNumber], hasNormalsPredBits);
      if (animatedFields & 4)
         FAMCCabacVx3Decoder
            resColors(numberOfVerticesInLayer[layerNumber], hasColorsPredBits);
      if (animatedFields & 8)
         FAMCCabacVx3Decoder resOtherAttributes(numberOfVerticesInLayer[layerNumber],
hasOtherAttributesPredBits);
      }
   }
}
```

4.4.2.16.2 Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For each frame (frameNumberDec) a frame header is decoded. Thereby the following values are decoded or computed: frameType, frameNumberDis, refFrameNumberOffsetDis0, refFrameNumberOffsetDis1, coordsPredictionModeLD, normalsPredictionModeLD, colorsPredictionModeLD, and otherAttributesPredictionModeLD.

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

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

Table 60 — The correspondences between predictionModeLD values and prediction types predType

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

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

If coordPredictionModeLD=0,1,2 then coordsPredType[frameNumberDis][layerNumber][c]=coordPredictionModeLD for all c and layers of a frame with frame number frameNumberDis.

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

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

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

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

resNormals[][], resColors[][], and resOtherAttributes[][] get prediction types normalsPredType[][], colorsPredType [][], and otherAttributesPredType [][] assigned).

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

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

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

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

4.4.2.17 FAMCLayeredDecompositionDecoder class

4.4.2.17.1 Syntax

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

4.4.2.17.2 Semantics

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

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

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

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

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

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

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

```
struct SimplificationOperation{
    int vertexIndex;
    int mode;
};

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

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

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

for layer=1,…,L-1 and c=0,…, numberOfVerticesInLayer[layer].are obtained. The procedure for building the layer decomposition from simplification operations is described in Annex I.

4.4.2.18 FAMCSimplificationModeDecoder class

4.4.2.18.1 Syntax

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

4.4.2.18.2 Semantics

simplificationMode: an arithmetic encoded integer indicating a simplification mode.

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

4.4.2.19 FAMCLDFrameHeaderDecoder class

4.4.2.19.1 Syntax

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

```
   unsigned int(4) colorsPredictionModeLD;
   unsigned int(4) otherAttributesPredictionModeLD;
}
```

4.4.2.19.2 Semantics

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

frameNumberDis=frameNumberPrevDis+frameNumberOffsetDis,

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

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

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

```
refFrameNumberDec0=frameNumberDec – refFrameNumberOffsetDec0.
```

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

refFrameNumberDec1=frameNumberDec – refFrameNumberOffsetDec1.

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

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

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

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

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

```
frameNumberDec2DisList[frameNumberDec]=frameNumberDis
```

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

4.4.2.20 FAMCCabacVx3Decoder

4.4.2.20.1 Syntax

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

   //decoding signs
   for (int v = 0; v < V; v++){
      for (int d = 0; d < 3; d++){
```

```
            if (sigMap[v][d] == 1) {
               negative[v][d] = cabac.biari_decode_symbol_eq_prob(cabac._dep);
            }
         }
      }

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

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

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

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

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

4.4.2.20.2 Semantics

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

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

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

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

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

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

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

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

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

4.5 Generic tools

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

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

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

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

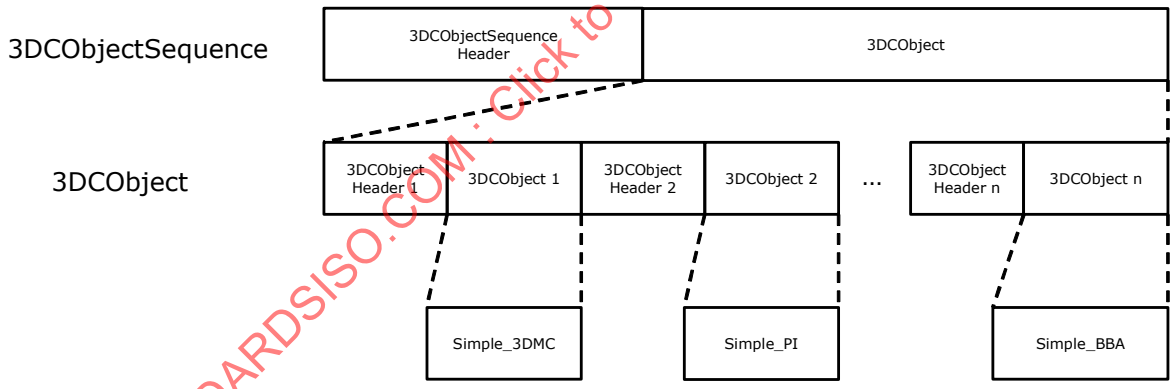


Figure 77 — Syntax of the MP43D stream

4.5.1.1 3DCObjectSequence

4.5.1.1.1 Syntax

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

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

4.5.1.1.2 Semantics

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

profile_and_level_indication: This is an 8-bit integer used to signal the profile and level identification. The meaning of the bits is given in Table 61.

Table 61 — FLC table for profile_and_level_indication

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

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

4.5.1.2 3DCObject

4.5.1.2.1 Syntax

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

   bit(32) *next;
   while (next == user_start_code) {
user_data();
   }
   bit(32) *next;
   if (next == "Simple_3DMC") {
      bit(32) simple_3dc_object_type_start_code;
   3D_Mesh_Object() ;
}
   else if (next == "Simple_WSS") {
      bit(32) simple_WSS_object_type_start_code;
   Wavelet_Mesh_Object();
}
   else if (next == "Simple_CI") {
      bit(32) simple_CI_object_type_start_code;
   CompressedCoordinateInterpolator() ;
```

```
    }
    else if (next == "Simple_OI") {
       bit(32) simple_OI_object_type_start_code;
       CompressedOrientationInterpolator () ;
    }
    else if (next == "Simple_PI") {
       bit(32) simple_PI_object_type_start_code;
       CompressedPositionInterpolator ();
    }
    else if (next == "Simple_BBA") {
       bit(32) simple_BBA_type_start_code;
       bba_object ();
    }
}
```

4.5.1.2.2 Semantics

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

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

3dc_object_verid: This is a 4-bit code which identifies the version number of the 3D Compression object. Its meaning is defined in Table 62. When this field does not exist, the value of 3dc_object_verid is '0001'.

Table 62 — Meaning of 3dc_object_verid

3dc_object_verid	Meaning
0000	reserved
0001	Object Types listed in Table 63
0010-1111	reserved

Table 63 list the tools included in each of the Object Types. The current object types can be extended when new tools or functionalities will be introduced.

Table 63 — Tools for 3D Compression Object Types

AFX Tools	3D Compression Object Types					
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Simple WSS	Simple BBA
3D Mesh Compression (3DMC) - Basic	X					
CoordinateInterpolator (CI)		X				
PositionInterpolator (PI) - Key Preserving - Path Preserving			X			

OrientationInterpolator (OI) - Key Preserving - Path Preserving				X		
Wavelet Subdivision Surface (WSS) - IndexedFaceSet for base mesh					X	
BBA - Only Bones						X

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

3dc_object_type_start_code: It is the bit string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' in binary for resynchronization. The last 8 bits represent the one of the values in the 'A6' to 'AB' to indicated object types defined in Table 64. According to the last 8 bits in **"3dc_object_type_start_code"**, the corresponding decoder is called and the compressed stream is decoded. If more object types are defined in Table 63, the added object types are reflected in Table 64.

Table 64 — Meaning of start code value

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

4.5.1.3 user_data

4.5.1.3.1 Syntax

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

4.5.1.3.2 Semantics

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

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

4.5.2 AFX Generic Backchannel

4.5.2.1 Usage of the backchannel

The backchannel can be used by the following tools : VTC textures, WaveletSubdivisionSurfaces, MeshGrid.

In the first case, i.e. VTC textures, the backchannel will be an elementary stream of stream-type 'visual' (0x04) and with an ObjectTypeIndication 0x20.

In the case of WaveletSubdivisionSurfaces and MeshGrid, the backchannel is an elementary stream of stream type 'scene description' (0x03) with an OTI 0x05. The AFXConfig contains the appropriate code to make the distinction between AFX tools (see ISO/IEC 14496-1 Amendment 4).

Each backchannel is related to a downstream elementary stream carrying the data associated with each tool.

This means that AFX tools share the same bitstream syntax but each instance has a separate elementary stream.

The messages carried in an AccessUnit of a backchannel consist of a list of BackChannelCommands as specified below. The syntax of an access unit is defined as follows.

4.5.2.2 BackChannel Access Unit

4.5.2.2.1 Syntax

```
class BackChannelAccessUnit {
   BackChannelCommand[0..255] backchannelCommands;
}
```

4.5.2.2.2 Semantics

An access unit in the backchannel is an array of BackChannelCommands.

4.5.2.3 BackChannelCommand

4.5.2.3.1 Syntax

```
abstract aligned(8) expandable(228-1) class BackChannelCommand : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

4.5.2.3.2 Semantics

The possible values of BackChannelCommand tag and their interpretation is given in Table 65.

Table 65 — List of Class Tags for Backchannel Commands

Tag value	Tag name
0x00	Forbidden
0x01	GlobalParametersTag
0x02	NavigationParametersTag
0x03	ToolGenericBackChannelTag
0x04-0xBF	Reserved for ISO (backchannel command tags)
0xC0-0xFE	User private
0xFF	Forbidden

4.5.2.3.3 General bitstream syntax

The bitstream is an array of BackChannelCommands.

The server decodes all backchannelcommands of the access unit which it can parse, otherwise it skips them.

The configuration of the stream describes the list of tags are mandatory in the bitstream as well as those that are optional (i.e. that a server may parse). Other backchannel commands will be skipped by a compliant server.

4.5.2.3.4 Definitions

4.5.2.3.4.1 Client/Server-driven scenario's

When considering view-dependent 3D transmission, two possible scenarios occur. Either the client tracks the cache status and asks the server for the not-yet-transmitted information, either it is the server that gets the client's cache status back and calculates out of this which additional information to transmit for the current viewpoint. Said in other words: the control about what to transmit and what is already available is either done by the client (first scenario), either by the server (second scenario). The scenarios are therefore called respectively *client-driven* and *server-driven*.

In a *client-driven* scenario, neither the viewpoint, nor the cache status have to be transmitted, since this information is used by the client only.

In the *server-driven* scenario, both the viewpoint and cache status have to be transmitted to the server.

4.5.2.3.4.2 Unit

A *unit* is a generic term that for each tool may have one or several meanings depending on the context as follows:

For wavelet subdivision surface: units represent hierarchical trees corresponding to small regions of the base mesh.

For VTC two contexts exist:

- A unit represents a Texture Unit that can be extracted from a packet in VTC's error resilience mode.

- A unit represents an ROI corresponding to a block in the spatial domain of size $2^{levels}x^{2levels}$ pixels. In the wavelet domain this corresponds to 9 trees (for each Y, U and V color component: the low-high, high-low and high-high subimage tree) or 3 trees (when only the Y component applies). The numbering of these ROIs is row-wise, starting with number 1 as shown in the following picture:

ROI nr 1	ROI nr 2	ROI nr 3	ROI nr 4
ROI nr 5	ROI nr 6	ROI nr 7	ROI nr 8
ROI nr 9	ROI nr 10	ROI nr 11	ROI nr 12
ROI nr 13	ROI nr 14	ROI nr 15	ROI nr 16

For MeshGrid two contexts are available:

- In mesh related commands (MESH_HEADER , MESH_CONNECTIVITY, VERTICES_REPOSITION and VERTICES_REFINEMENT) a unit represents a region of interest (ROI).
- In grid related commands (GRID_HEADER and GRID_COEFFICIENTS) a unit represents a tile.

4.5.2.3.5 GlobalParameters

4.5.2.3.5.1 Syntax

```
class GlobalParameters extends BackChannelCommand:
bit(8) tag = GlobalParametersTag {
   SFVec2f FieldOfView;
   SFFloat NearPlane;
   SFFloat FarPlane;
   SFFloat VisibilityLimit;
}
```

4.5.2.3.5.2 Semantics

Field of view: these are the horizontal and vertical angles of the view pyramid.

NearPlane: this is the distance to the *near plane* of the viewing pyramid, i.e. a plane parallel to the projection plane and at distance **NearPlane** to the observer. Every point on the same side as the observer is supposed to be non visible.

FarPlane: this is the distance to the *far plane* of the viewing pyramid, i.e. a plane parallel to the projection plane and at distance **FarPlane** to the observer. Every point beyond is non visible.

VisibilityLimit: this is a float indicating a distance to the observer beyond which the scene does not have to be displayed. This might be useful, e.g. for simulating fog.

4.5.2.3.6 NavigationParameters

4.5.2.3.6.1 Syntax

```
class NavigationParameters extends BackChannelCommand:
```

```
bit(8) tag = NavigationParametersTag {
   SFVec3f Position;
   SFRotation Direction;
}
```

4.5.2.3.7 Semantics

Position: this is the observer position in the coordinate system of the node corresponding to the transmitted object.

Direction: this is the observer orientation in the coordinate system of the node corresponding to the transmitted object.

4.5.2.3.8 ToolGenericBackChannel

4.5.2.3.8.1 Syntax

```
class ToolGenericBackChannel extends BackChannelCommand:
bit(8) tag = ToolGenericBackChannelTag {
   bit (1) isClientDriven;
   if (!isClientDriven) {
      bit (2) cacheMode;
   }
   ListOfUnits listOfUnits[];
}
```

4.5.2.3.8.2 Semantics

isClientDriven: 1-bit flag specifying if equal to '1' that the backchannel is in *client-driven* scenario and conveys client requests. If the value is equal to '0' the backchannel is in *server-driven* scenario and the client sends status information to the server. The requests and status information are specific for each tool extending the generic ToolGenericBackChannel class.

cacheMode: 2-bit value specifying the type and format of the status information send in the *server-driven* scenario. The following table shows the possible values and meanings of cacheMode. In **DELTA_CACHE** mode the status information contains details about received units that have been totally or partially discarded. Both the **RECEIVED_CACHE** and **FULL_CACHE** modes reflect the same type of cache information, however using a different format. In **RECEIVED_CACHE** mode the units are explicitly identified by an index. The status information expressed using this format is compact if consecutive units in the clients cache have the same "quality" parameters and thus can be grouped together by means of **ParseIndices** class. In the opposite case when consecutive units in the client's cache have different "quality" parameters, it is more efficient to send the status information in **FULL_CACHE** mode by iterating and querying the entire range of units.

cacheMode	Meaning
0x0	DELTA_CACHE: Delta cache status, i.e. received units that have been partially or totally discarded.
0x1	RECEIVED_CACHE: Cache status, i.e. information only about the received units.
0x2	FULL_CACHE: Full cache status, i.e. information about the entire range of units.
0x3	Reserved

listOfUnits: Specifies an array of commands that have to be processed simultaneously, in order to have a consistent decoding process at the client. In particular, if a Region of Interest (ROI) has to be decoded at a given quality, all data that contributes to this ROI (through data dependencies) should be decoded properly.

This may require several, successive commands, each pertaining to specific information to be decoded at a specific quality.

4.5.2.3.9 ListOfUnits

4.5.2.3.9.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class ListOfUnits (bool isClientDriven, unsigned int
cacheMode) : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

4.5.2.3.9.2 Semantics

This is an abstract base class for the different types of command units in the backchannel stream. This class is extended by the classes identified by the class tags defined in the following table.

Table 66 — List of Class Tags for Descriptors derived from ListOfUnits

Tag value	Tag name	Description
0x00	Forbidden	
0x01	ListOfVTCUnitsTag	Tag for VTC units
0x02	ListOfWaveletTreesTag	Tag for WaveletSubdivisionSurface units
0x03	MGFullStreamTag	Tag for the entire MeshGrid stream.
0x04	MGMeshDescrTag	MeshGrid stream tag for the mesh coding information at specified mesh resolution level.
0x05	MGGridDescrTag	MeshGrid stream tag for grid coding information at specified grid resolution level.
0x06	MGMeshConnectivityDescrTag	MeshGrid stream tag for mesh connectivity information at specified mesh resolution level.
0x07	MGVerticesRepositionDescrTag	MeshGrid stream tag for vertices' reposition bits (single bit-plane) at specified mesh resolution level.
0x08	MGVerticesRefinementDescrTag	MeshGrid stream tag for refinement bit-planes (the offset) at specified mesh resolution level.
0x09	MGGridCoefficientsDescrTag	MeshGrid stream tag for wavelet coefficients at specified grid resolution level.
0x10	MGGridCornersDescrTag	MeshGrid stream tag for the grid corners.
0x11-0xFE	Reserved for ISO use	
0xFF	Forbidden	

4.5.2.3.10 GetValue

4.5.2.3.10.1 Syntax

```
class GetValue(BYTE nBits)
{
   int (nBits) noBits;
   int (noBits) value;
}
```

4.5.2.3.10.2 Semantics

The **GetValue** class parses a variable length integer *value*. The result is returned in *value*.

4.5.2.3.11 ParseIndices

4.5.2.3.11.1 Syntax

```
class ParseIndices(BYTE nBits) {
   unsigned int i = 0, j = 0, k = 0;

   // the number of elements in the indices list.
   GetValue number(nBits);

   for (i = 0; i < number.value; i++) {
   bit (1) isIndexed;
      if (isIndexed) {
         GetValue unit[[k++]](nBits);    // index of the unit
      } else {
         GetValue start(nBits);          // start index of the unit
         GetValue count(nBits);          // number of indices in the range
         for (j = 0; j < count.value; j++) {
         unit[[k++]] = start.value + j;
         }
      }
   }

    // assign the total number of units.
   number.value = k;
}
```

4.5.2.3.11.2 Semantics

Parse a list of units that can be read one by one with an index and/or units that can be read as a successive range.

number: The initial parsed value represents the number of elements in the indices list. A list element can be the index of a unit (**unit**), or the start index (**start**) of a range of indices followed by a number (**count**) specifying the number of indices in the range. The final value of number is to the total number of units from the **unit** array.

isIndexed: Flag, for which if equal to '1' specifies that the following list element is an unit index; otherwise it specifies that the following two values represent the start index of a unit (**start**) and the number of indices in the range (**count**).

unit: Array of size **number** consisting of indices of units.

4.5.2.3.12 ListOfVTCUnits

4.5.2.3.12.1 Syntax

```
class ListOfVTCUnits extends ListOfUnits(bool isClientDriven, unsigned int cacheMode):
bit(8) tag = ListOfVTCUnitsTag {
   bit (1) isROI;
   int i,j = 0;
   if (isROI)
   {
      if (isClientDriven) {
         int (5) firstWantedLevel;
         int (5) lastWantedLevel;
         int (5) firstWantedBitplane;
         int (5) lastWantedBitplane;
         ParseIndices ROI(5);
      } else if (cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
         int (5) lastCachedLevel;
         int (5) lastCachedBitplane;
         ParseIndices ROI(5);
      } else {                  // server-driven & cacheMode == FULL_CACHE
         GetValue numberOfROIs(5);
         for (j=0; j<numberOfROIs.value; j++) {
            bit(1) isInCache;
            if (isInCache) {
               int (5) lastCachedLevel;
               int (5) lastCachedBitplane;
            }
         }
      }
   }
   else
   {
      if (isClientDriven || cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
         ParseIndices TU(5);
      } else {                // server-driven & cacheMode == FULL_CACHE
         GetValue numberOfTUs(5);
         for (j=0; j<numberOfTUs.value; j++) {
            bit(1) isInCache;
         }
      }
   }
}
```

4.5.2.3.12.2 Semantics

In VTC, the quality of decoded texture regions can be set by selecting the number of decoded wavelet levels and bitplanes.

isROI: this bit indicates the type of units used by the command. If true, ROI numbers are used with specified wavelet levels and bitplanes. If false, texture units (TU) are used. The numbering of the TUs implicitly and uniquely indicates the corresponding texture region location in spatial domain at a certain spatial/resolution layer (i.e. number of wavelet levels) and SNR layer (i.e. number of bitplanes), therefore wavelet levels and bitplanes do not have to be specified.

In the *client-driven* scenario and when isROI is true, several ROIs with the same “quality” requests (**firstWantedLevel**, **lastWantedLevel, firstWantedBitplane, lastWantedBitplane**) are grouped together in one command. When isROI is false, TUs are grouped together in one command.

firstWantedLevel, **lastWantedLevel**: these are the wavelet decomposition levels (ranging from first to last) that the client asks for, in order to increase the resolution-driven quality from previously cached information (which is equivalent to lastCachedLevel).

firstWantedBitplane, lastWantedBitplane: these are the bitplanes (ranging from first to last) that the client asks for, in order to increase the SNR-driven quality from previously cached information (which is equivalent to lastCachedBitplane).

In the *server-driven* scenario when the cache mode is DELTA_CACHE or RECEIVED_CACHE and when isROI is true, several ROIs with the same "quality" status (**lastCachedLevel, lastCachedBitplane**) are grouped together in a command. When isROI is false, TUs are grouped together in one command.

lastCachedLevel: this is the last wavelet decomposition level that is cached at the client side.

lastCachedBitplane: this is the last bitplane that is cached at the client side.

In the *server-driven* scenario when the cache mode is FULL_CACHE and isROI is true, iterate for all ROIs (**numberOfROIs**). For each ROI, the **isInCache** flag specifies with '1' that cache status information (**lastCachedLevel, lastCachedBitplane**) is available; conversely if **isInCache** flag is '0' then the corresponding ROI is not present in the cache. When isROI is false iterate for all TUs. For each TU, the **isInCache** flag specifies with '1' that the TU is available; conversely if **isInCache** flag is '0' then the corresponding TU is not present in the cache.

4.5.2.3.13 ListOfWaveletTrees

4.5.2.3.13.1 Syntax

```
class ListOfWaveletTrees extends ListOfUnits(bool isClientDriven, unsigned int cacheMode):
bit(8) tag = ListOfWaveletTreesTag {
   int i = 0;
   if (isClientDriven) {
      GetValue numWaveletCoefficients(5);
      int(5) firstWantedBitplane;
      int(5) lastWantedBitplane;
      ParseIndices tree(5);
   } else if (cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
      GetValue numWaveletCoefficients(5);
      int(5) lastCachedBitPlane;
      ParseIndices tree(5);
   } else {                    // server-driven & cacheMode == FULL_CACHE
      GetValue numberOfTrees(5);
      for (i=0; i<numberOfTrees.value; i++) {
         bit(1) isInCache;
         if (isInCache) {
            GetValue numWaveletCoefficients(5);
            int(5) lastCachedBitPlane;
         }
      }
   }
}
```

4.5.2.3.13.2 Semantics

The units for WaveletSubdivisionSurfaces are trees used in the SPIHT representation of the wavelet coefficients.

NumWaveletCoefficients: this is the number of wavelet coefficients. If **isClientDriven** value is 1, **numWaveletCoefficients** is understood as the number of coefficients the client needs for tree number i. If **isClientDriven** value is 0, **numWaveletCoefficients** is understood as the number of coefficients that the client cache contains for ROI number i (if **isClientDriven** value is 1) or number **TreeNb** (if **isClientDriven** value is 0 and **isCacheDelta** value is 1).

firstWantedBitplane: this is the number of the first requested bitplane.

lastWantedBitplane: this is the number of the last requested bitplane.

lastCachedBitPlane: if **cacheMode** value is DELTA_CACHE or RECEIVED_CACHE, this is the number of bitplanes present in the cache for each tree contained in tree; if **cacheMode** value is FULL_CACHE, this is the number of bitplanes present in the cache for tree number i.

tree: if **isClientDriven** has value 1, this is the sequence of trees to which **NumWaveletCoefficients**, **firstWantedBitplane** and **lastWantedBitplane** refer. Otherwise this is the sequence of trees to which **NumWaveletCoefficients** and **lastCachedBitplane** refer.

NumberOfTrees: this is the total number of trees for the object.

IsInCache: this is a flag with value 1 if and only if cache information for tree number i is following.

4.5.2.3.14 MGFullStreamCommand

4.5.2.3.14.1 Syntax

```
class MGFullStreamCommand extends ListOfUnits(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGFullStreamTag {
   // empty
}
```

4.5.2.3.14.2 Semantics

For both *client-driven* and *server-driven* scenarios request the entire stream in one step. This command is equivalent with grouping in one request several commands issued for each resolution level of the mesh, respectively grid, but it is far more compact.

4.5.2.3.15 MGLevelCommand

4.5.2.3.15.1 Syntax

```
abstract class MGLevelCommand extends ListOfUnits(bool isClientDriven, unsigned int
cacheMode): bit(8) tag=0
{
   // read the variable length counter sizeOfInstance
   unsigned int(LEVEL_BITS) resolutionLevel;
}
```

4.5.2.3.15.2 Semantics

This is an abstract class that serves as a base class for MeshGrid related command units. MGLevelCommand parses the resolution level (*resolutionLevel*) of the unit.

4.5.2.3.16 MGLayerCommand

4.5.2.3.16.1 Syntax

```
abstract class MGLayerCommand extends MGLevelCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag=0
{
   // read the variable length counter sizeOfInstance
   ParseIndices identifier(nBitsLayer);
}
```

4.5.2.3.16.2 Semantics

This is an abstract class that serves as a base class for MeshGrid related command units. MGLayerCommand parses an identifier of the surface layers the MeshGrid stream unit refers to. When the value of identifier.number is ‘0’ then the request is is generic for all surface layers.

4.5.2.3.17 MGMeshDescriptorCommand

4.5.2.3.17.1 Syntax

```
class MGMeshDescriptorCommand extends MGLayerCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGMeshDescrTag {
   // empty
}
```

4.5.2.3.17.2 Semantics

For both *client-driven* and *server-driven* scenarios request mesh coding information for a specified mesh resolution level (**resolutionLevel**).

4.5.2.3.18 MGGridDescriptorCommand

4.5.2.3.18.1 Syntax

```
class MGGridDescriptorCommand extends MGLevelCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGGridDescrTag {
   // empty
}
```

4.5.2.3.18.2 Semantics

For both *client-driven* and *server-driven* scenarios request grid coding information at a specified grid resolution level (**resolutionLevel**).

4.5.2.3.19 MGMeshConnectivityCommand

4.5.2.3.19.1 Syntax

```
class MGMeshConnectivityCommand extends MGLayerCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGMeshConnectivityDescrTag {
   if (isClientDriven ||
       cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
      ParseIndices roi(ROI_BITS);
   } else {                  // server-driven & cacheMode == FULL_CACHE
      GetValue numberOfROIs(ROI_BITS);
      for (j = 0; j < numberOfROIs.value; j++) {
         bit(1) isInCache;
      }
   }
}
```

4.5.2.3.19.2 Semantics

The mesh connectivity information for specified regions of interest (ROIs) and mesh resolution level (**resolutionLevel**) is requested in *client-driven* mode, respectively the corresponding cache status is sent in *server-driven* mode. The ROIs are given explicitly in client-driven scenario, or in server-driven scenarios with DELTA_CACHE and RECEIVED_CACHE operation modes. In server-driven scenario FULL_CACHE operation mode all possible ROIs (**numberOfROIs**) are iterated, and for each ROI the **isInCache** flag specifies if '1' that the connectivity description is available at the client site, and if '0' it indicates the opposite.

4.5.2.3.20 MGVerticesRepositionCommand

4.5.2.3.20.1 Syntax

```
class MGVerticesRepositionCommand extends MGLayerCommand(bool isClientDriven, unsigned int
cacheMode) : bit(8) tag= MGVerticesRepositionDescrTag {
   if (isClientDriven ||
       cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
      ParseIndices roi(ROI_BITS);
   } else {                // server-driven & cacheMode == FULL_CACHE
      GetValue numberOfROIs(ROI_BITS);
      for (j=0; j<numberOfROIs.value; j++) {
         bit(1) isInCache;
      }
   }
}
```

4.5.2.3.20.2 Semantics

The vertices' reposition bit-plane for specified regions of interest (ROIs) and mesh resolution level (**resolutionLevel**) is requested in *client-driven* mode, respectively the corresponding cache status is sent in *server-driven* mode. The ROIs are given explicitly in client-driven scenario, or in server-driven scenarios with DELTA_CACHE and RECEIVED_CACHE operation modes. In server-driven scenario FULL_CACHE operation mode all possible ROIs (**numberOfROIs**) are iterated, and for each ROI the **isInCache** flag specifies if '1' that the vertices' reposition bit-plane is available at the client site, and if '0' it indicates the opposite.

4.5.2.3.21 MGVerticesRefinementCommand

4.5.2.3.21.1 Syntax

```
class MGVerticesRefinementCommand extends MGLayerCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGVerticesRefinementDescrTag {
   if (isClientDriven) {
      bit(REFINE_BITS) firstWantedBitPlane;
      bit(REFINE_BITS) lastWantedBitPlane;
      ParseIndices roi(ROI_BITS);
   } else if (cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
      bit(REFINE_BITS) lastCachedBitPlane;
      ParseIndices roi(ROI_BITS);
   } else {                // server-driven & cacheMode == FULL_CACHE
      GetValue numberOfROIs(ROI_BITS);
      for (j=0; j<numberOfROIs.value; j++) {
         bit(1) isInCache;
         if (isInCache) {
            bit(REFINE_BITS) lastCachedBitPlane;
         }
      }
   }
}
```

4.5.2.3.21.2 Semantics

For specified regions of interest (ROIs) and mesh resolution level (**resolutionLevel**), the vertices' refinement bit-planes, starting with **firstWantedBitPlane** and ending with **lastWantedBitPlane,** are requested in *client-driven* scenario, respectively the corresponding last bit-plane available in the cache (**lastCachedBitPlane**) is sent in the *server-driven* scenario. The ROIs are given explicitly in client-driven scenario, or in server-driven scenarios with DELTA_CACHE and RECEIVED_CACHE operation modes. In server-driven scenario FULL_CACHE operation mode all possible ROIs (**numberOfROIs**) are iterated, and for each ROI the **isInCache** flag specifies if '1' that the connectivity description is available at the client site, and if '0' it indicates the opposite.

4.5.2.3.22 MGGridCoefficientsCommand

4.5.2.3.22.1 Syntax

```
class MGGridCoefficientsCommand extends MGLevelCommand(bool isClientDriven, unsigned int
cacheMode): bit(8) tag= MGGridCoefficientsDescrTag {
   if (isClientDriven) {
      bit(FIELD_BITS) firstWantedBitPlane;
      bit(FIELD_BITS) lastWantedBitPlane;
      ParseIndices tile(ROI_BITS);
   } else if (cacheMode == DELTA_CACHE || cacheMode == RECEIVED_CACHE) {
      bit(FIELD_BITS) lastCachedBitPlane;
      ParseIndices tile(ROI_BITS);
   } else {                // server-driven & cacheMode == FULL_CACHE
      GetValue numberOfTiles(ROI_BITS);
      for (j = 0; j < numberOfTiles.value; j++) {
         bit(1) isInCache;
         if (isInCache) {
            bit(FIELD_BITS) lastCachedBitPlane;
         }
      }
   }
}
```

4.5.2.3.22.2 Semantics

At any grid resolution level (**resolutionLevel**) for any encoded tile (**tile**, **numberOfTiles**) there are three binary streams, called *component streams*, each of them corresponding to one of the {*x*,*y*,*z*} wavelet encoded coordinates of the grid points. In *client-driven* scenario, the grid coding bitplanes, starting with **firstWantedBitPlane** and ending with **lastWantedBitPlane,** are request to the server. In *server-driven* scenario, the client sends the last bit-plane available in the cache (**lastCachedBitPlane**). In *client-driven* scenario, and in *server-driven* scenarios with DELTA_CACHE and RECEIVED_CACHE operating modes, the tiles (**tile**) are explicitly specified using ParseIndices. In *server-driven* scenario FULL_CACHE operation mode, all the tiles (**numberOfTiles**) are iterated, and for each tile the **isInCache** flag specifies if '1' that cache information is available at the client site, and if '0' it indicates the opposite.

4.5.2.3.23 MGGridCornersCommand

4.5.2.3.23.1 Syntax

```
class MGGridCornersCommand extends ListOfUnits(bool isClientDriven, unsigned int
cacheMode): bit(8) tag=MGGridCornersDescrTag
{
   // empty
}
```

4.5.2.3.23.2 Semantics

In client-driven scenario request, respectively in server-driven scenario send, the coordinates of the reference-grid corners.

5 AFX object codes

AFXExtDescriptor described in ISO/IEC 14496-1 is an abstract class used as a placeholder for an optional DecoderSpecificInfo defined in Table 67.

Table 67 — DecoderSpecificInfo for AFX streams

AFX stream	DecoderSpecificInfo
MeshGrid	See subclause 5.2.2.2.
WaveletSubdivisionSurface	See subclause 5.1.1.1.
Other AFX streams	None

The **tag** field in the **AFXExtDescriptor** refers to a specific node compression scheme as defined in Table 28. Each node compression scheme is used to decode the bistream composed of associated node. If associated node is same and bitwrapper is used in buffer scenario, the **type** value of bitwrapper is needed.

Table 68 — AFX object code

AFX object code	Object	Associated node	Type value of bitwrapper
0x00	3D Mesh Compression	IndexedFaceSet	0
0x01	WaveletSubdivisionSurface	WaveletSubdivisionSurface	0
0x02	MeshGrid	MeshGrid	0
0x03	CoordinateInterpolator	CoordinateInterpolator	0
0x04	OrientationInterpolator	OrientationInterpolator	0
0x05	PositionInterpolator	PositionInterpolator	0
0x06	OctreeImage	OctreeImage	0
0x07	BBA	SBVCAnimation	0
0x08	PointTexture	PointTexture	0
0x09	3D Mesh Compression Extension	IndexedFaceSet	1
0x0A	FootPrint Based Representation	FootprintSetNode	0
0x0B	FrameBasedAnimatedMeshCompression	-	-

6 3D Graphics Profiles

6.1 Introduction

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

6.2 "Graphics" Dimension

6.2.1 MPEG-4 X3D Interactive Graphics Profiles and Levels

6.2.1.1 List of tools/functionalities

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

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

6.2.1.2 Comparison with existing profiles

The X3D Interactive Graphics profile is based on X3D level 1 Interactive Profile [95], and adds the following MPEG-4 features: BIFS-commands for streaming and Quantization for compression efficiency. No other profile in MPEG-4 addresses 3D (only) environments.

6.2.1.3 X3D Interactive Graphics Profile @ Level 1 Definition

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

Table 69 — Level 1 of Web3D Interactive Graphics profile

Node	Minimum System Support
Appearance	textureTransform not supported.
Background	groundAngle and groundColor not supported.
	texture treated as background image FALSE. stretchToFit not supported.
	backURL, frontURL, leftURL, rightURL, topURL are not supported.
	One skyColor.
Box	Full support
Color	15,000 colors.
Cone	Full support
Coordinate	65,535 points.
Cylinder	Full support
Directional Light	AmbientIntensity not supported.
	Not scoped by parent Group or Transform.

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

PointSet	5000 points.
Shape	Full support.
Sphere	Full support
SpotLight	Ignore beamWidth. Ignore radius.
	Ignore ambientIntensity.
	Linear attenuation.
TextureCoordinate	65,535 coordinates.

Table 70 specifies further restriction to the fields of the nodes listed in Table 69. These tables can be used for both the Profile and the Level definitions.

Table 70 — Functionality limitation and minimum system requirement

Node	Restrictions
All lights	8 simultaneous lights.
Names for DEF/PROTO/field	50 utf8 octets.
All url fields	10 URLs.
SFBool	Full support.
SFColor	Full support.
SFFloat	Full support.
SFImage	256 width. 256 height.
SFInt32	Full support.
SFNode	Full support.
SFRotation	Full support.
SFString	30,000 utf8 octets.
SFTime	Full support.
SFVec2d	15,000 values.
SFVec2f	15,000 values.
SFVec3d	15,000 values.
SFVec3f	15,000 values.
MFColor	15,000 values.
MFFloat	1,000 values.

MFInt32	20,000 values.
MFNode	500 values.
MFRotation	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.
MFVec2d	15,000 values.
MFVec2f	15,000 values.
MFVec3d	15,000 values.
MFVec3f	15,000 values.

NOTE: The X3D interactive Profile is a common compatibility point with We3D's X3D Interactive Profile. The following tools are supported in MPEG, but not in Web3D's profile: QuantizationParameter, Node Update, Scene Update and Route Update.

6.2.2 MPEG-4 "Basic AFX" Graphics Profiles and Levels

6.2.2.1 List of tools/functionalities

The Basic AFX Graphics Profile represents a collection of nodes to allow progressive and adaptive transmission over networks of large 3D environments and / or complexe 3D shapes. It includes the following nodes : Appearance, Background, Color, Coordinate, DirectionalLight, ElevationGrid, IndexedFaceSet, IndexedLineSet, Material, PointLight, Shape, SpotLight, TextureCoordinate, TextureTransform, ProceduralTexture (V.5), SBVCAnimation, SBVCSkinnedModel, SBBone, SBSegment, SubdivisionSurfaces, WaveletSubdivisionSurfaces and FootPrint.

6.2.2.2 Comparison with existing profiles

The Basic AFX Scene Graph profile represents a collection of nodes to allow progressive and adaptive transmission over networks of large 3D environments and / or complexe 3D shapes.

The existing 'X3D' profile does not provide compression tools.

6.2.2.3 Basic AFX Graphics Profile @ Level 1 and 2 Definition

In the following table, definitions for level 1 of the 3D Multiresolution Graphics profile are given.

Table 71 — Level 1 & 2 of Basic AFX Graphics profile

Node	Maximum values for content related parameters	
	Level 1	**Level 2**
Appearance	Ignore TextureTransform.	Full features supported.
Color	216 colors*	232 colors*
Coordinate	216 points*	232 points*
DirectionalLight	Not scoped by parent Group or Transform.	Scoped by parent Group or Transform.

IndexedFaceSet	Only triangle face supported. A given coordIndex is not repeated in a face. Ignore set_colorIndex. Ignore set_normalIndex.	Full features supported.
IndexedLineSet	Ignore set_colorIndex. Ignore set_coordIndex.	Full features supported.
Material	Ignore AmbientIntensity. Ignore Shininess. Ignore SpecularColor.	Full features supported.
PointLight	Ignore radius. Ignore Linear attenuation.	Full features supported.
Shape	Full features support.	Full features supported.
SpotLight	Ignore beamWidth. Ignore radius. Ignore Linear attenuation.	Full features supported.
TextureCoordinate	216 coordinates*	232 coordinates*
WaveletSubdivision Surfaces	12 bitplanes per coordinate 4 levels of subdivision	24 bitplanes per coordinates 10 levels of subdivision
FootPrint	Full features supported.	Full features supported.

* indicates maximum vector size.

The following table specifies further restriction to the fields of the nodes listed above. These two Tables can be used for both the Profile and the Level definitions.

Table 72 — Functionality limitation and minimum system requirement

Node	Restrictions (Maximum values)
All lights	8 simultaneous lights.
Names for DEF/field	50 utf8 octets.
All url fields	10 URLs. URN's ignored. Support relative URLs where relevant.
SFBool	Full support.

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

6.3 "Scene Graph" Dimension

6.3.1 MPEG-4 X3D Interactive Scene Graph Profile and Levels

6.3.1.1 List of tools/functionalities

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

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

6.3.1.2 BIFS nodes for support of Audio and Visual objects

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

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

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

6.3.1.3 Comparison with existing profile

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

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

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

Table 73 — Level 1 of X3D Interactive Scene Graph profile

Node	Minimum Browser Support
Anchor	addChildren not supported.
	removeChildren not supported.
	Ignore parameter.
ColorInterpolator	Full support.
CoordinateInterpolator	15,000 coordinates per keyValue.
CylinderSensor	Full support.
Group	AddChildren not supported. removeChildren not supported.
Inline	Full support.
NavigationInfo	avatarSize not supported.
	speed not supported.
	Ignore visibilityLimit
OrientationInterpolator	Full support.
PlaneSensor	Full support.
PositionInterpolator	Full support.
ProximitySensor	Full support.
ScalarInterpolator	Full support.
SphereSensor	Full support.
Switch	Full support.

TimeSensor	Ignored if cycleInterval < 0.01 second.
TouchSensor	hitNormal_changed not supported
Transform	addChildren not supported. RemoveChildren not supported.
Viewpoint	Ignore fieldOfView.
WorldInfo	Full support
QuantizationParameter	Full support
Node Updates	Full support
Scene Updates	Full support
Route Updates	Full support

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

Table 74 specifies other aspects of functionality that are supported by this profile. Note that general items refer only to those specific nodes listed in Table 73.

Table 74 — Functionality Limitations and Minimum System Requirements

Node	Minimum System Support
All groups	500 children. Ignore bboxCenter and bboxSize.
All interpolators	1000 key-value pairs.
Names for DEF/PROTO/field	50 utf8 octets.
All url fields	10 URLs
SFBool	Full support.
SFFloat	Full support.
SFImage	256 width. 256 height.
SFInt32	Full support.
SFNode	Full support.
SFRotation	Full support.
SFString	30,000 utf8 octets.
SFTime	Full support.
SFVec2d	15,000 values.
SFVec2f	15,000 values.

SFVec3d	15,000 values.
SFVec3f	15,000 values.
MFFloat	1,000 values.
MFInt32	20,000 values.
MFNode	500 values.
MFRotation	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.
MFVec2d	15,000 values.
MFVec2f	15,000 values.
MFVec3d	15,000 values.
MFVec3f	15,000 values.

6.3.2 PEG-4 "Basic AFX" Scene Graph Profile and Levels

6.3.2.1 List of tools/functionalities

The Basic AFX Scene Graph profile represents a collection of nodes to allow progressive and adaptive transmission over networks of large 3D environments and / or complexe 3D shapes. It contains the same set of nodes as the X3D Scene Graph Profile, plus the Bitwrapper node.

6.3.2.2 Comparison with existing profiles

The Basic AFX Scene Graph profile represents a collection of nodes to allow progressive and adaptive transmission over networks of large 3D environments and / or complexe 3D shapes.

The existing 'X3D' profile does not provide compression tools.

6.3.2.3 "Basic AFX" Scene Graph Profile @ Level 1 Definition

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

Table 75 — Level 1 of Basic AFX Scene Graph profile

Node	Maximum values for content related parameters	
	Level 1	Level 2
CoordinateInterpolator	Full features supported.	Full features supported.
Group	Ignore AddChildren, Ignore removeChildren	Full features supported.
NavigationInfo	Ignore AvatarSize, Ignore speed,	Full features supported.

	Ignore type, Ignore visibilityLimit	
OrientationInterpolator	Full features supported.	Full features supported.
PositionInterpolator	Full features supported.	Full features supported.
ScalarInterpolator	Full features supported.	Full features supported.
TouchSensor	Full features supported.	Full features supported.
Transform	Ignore AddChildren, Ignore removeChildren	Full features supported.
Viewpoint	Ignore FieldOfView, Ignore description	Full features supported.
WorldInfo	Full features supported.	Full features supported.
QuantizationParameter	Full features supported.	Full features supported.
Scene Updates	Full features supported.	Full features supported.
ROUTE	Full features supported.	Full features supported.

The following table specifies other aspects of functionality that are supported by this profile. Note that general items refer only to those specific nodes listed in the table above.

Table 76 — Functionality Limitations and Minimum System Requirements

Node	Minimum System Support
All groups	500 children. Ignore bboxCenter and bboxSize.
All interpolators	1000 key-value pairs.
Names for DEF/field	50 utf8 octets.
All url fields	10 URLs. URN's ignored. Support relative URLs where relevant.
SFBool	Full support.
SFFloat	Full support.
SFImage	256 width. 256 height.
SFInt32	Full support.
SFNode	Full support.
SFRotation	Full support.
SFString	30,000 utf8 octets.

SFTime	Full support.
SFVec2f	15,000 values.
SFVec3d	15,000 values.
SFVec3f	15,000 values.
MFFloat	1,000 values.
MFInt32	20,000 values.
MFNode	500 values.
MFRotation	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.
MFVec2f	15,000 values.
MFVec3d	15,000 values.
MFVec3f	15,000 values.

6.4 "3D Compression" Dimension

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

3DCompressionProfileLevelIndication is defined in the following table.

Table 77 — 3DCompressionProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Core	L1
0x02	Core	L2
0X03	3D Multiresolution	L1
0X04	3D Multiresolution	L2
0x0A-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no 3D Compression profile specified	-
0xFF	no 3D Compression capability required	-
Note: Usage of the value 0xFE may indicate that the content described by this descriptor does not comply to any conformance point specified in this international standard.		

6.4.1 "Core 3D Compression" Profile and Levels

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

6.4.1.1 List of Tools/Functionalities

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

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

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

Table 78 — 3D Compression Object Types

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

BBA - Only Bones						X

The "Core 3D Compression" includes the object types as illustrated in Table 78.

Table 79 — "Core 3D Compression" Profile

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

6.4.1.2 Comparison with Existing Profiles and object types

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

6.4.1.3 Profile Level Definition

According to target device and applications, we defined two levels as listed in Table 80, Table 81 and Table 82. The level 1 is for the mobile device without H/W graphics accelerator. Thus, it is suitable simple application such as 3D Background, 3D GUI, 3D Pre-Viewer and 3D Avatar. On the other hand, the level 2 is for the mobile device supported by H/W graphics accelerator. Thus, the level 2 is suitable for the 3D Game application in addition to the application in level 1.

Table 80 — Levels and data constraints

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

Table 81 — Levels and functionalities constraints

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

Table 82 — Levels and player constraints

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

	- Texture Blending - Texture Filtration - Perspective correction - Total Decoding Time ≤ 10sec

6.4.2 3D Multiresolution Compression Profile and Levels

6.4.2.1 List of Tools/Functionalities

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

6.4.2.2 List of Tools/Functionalities

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

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

- The Simple 3DMC object type provides high compression and error resilience for static triangle 3D models.
- The Simple CI object type compresses the Coordinate Interpolator animation.
- The Simple PI object type compresses the Position Interpolator animation. It can support both Key-Preserving and Path-Preserving mode.
- The Simple OI object type compresses the Orientation Interpolator animation. It can support both Key-Preserving and Path-Preserving mode.
- The Main WSS object type represents, in a compressed form, the details for subdivision of 3D mesh. This tool is used for level of detail management and animation.
- The Simple FootPrint object type represents in a compressed form the multiresolution of 2D and a half data.
- The Simple BBA object type compresses the skeleton animation based on bone transforms and connected to a skin mesh model. This object type does not support Muscle.

Table 83 — 3D Compression Object Types

AFX Tools	3D Compression Object Types						
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Main WSS	Simple BBA	Simple Footprint
3D Mesh Compression (3DMC) Basic	X						
CoordinateInterpolator (CI)		X					
PositionInterpolator (PI) Key Preserving Path Preserving			X				
OrientationInterpolator (OI) Key Preserving Path Preserving				X			
avelet Subdivision Surface (WSS) IndexedFaceSet or 3DMC for base mesh Backchannel enabled					X		
BBA Only Bones						X	
Footprint-based Coding Bckchannel enabled							X

The "3D Multiresolution Compression" includes the object types as illustrated in the following table.

Table 84 — "3D Multiresolution Compression" Profile

	3D Compression Object Types						
	Simple 3DMC	Simple CI	Simple PI	Simple OI	Main WSS	Simple BBA	Simple Footprint
3D Multiresolution Compression Profile	x	x	x	x	x	x	x

6.4.2.3 Comparison with Existing Profiles and object types

The existing ‘Core 3D Profile’ is targeted for mobile applications and contains much simpler tools.

6.4.2.4 Profile Level Definition

According to target device and applications, two levels are defined as listed in the following tables. Level 1 is for mobile devices, while the level 2 is targeted at workstations or dedicated hardware.

Table 85 — Levels and data constraints

Level	Data Constraints	
Level 1	General	Number of triangles in a scene ≤ 5000 Number of objects in a scene ≤ 100 16.16 FixedPoint, 8-bit and 16-bit Integer
	BBA	Number of bones attachable to an object ≤ 30 Number of bones affecting a vertex ≤ 4
	WSS	Number of triangles in the base mesh ≤ 500 Number of bitplanes ≤ 10 Number of levels of details ≤ 4
	Footprint	Number of buildings ≤ 500 Tree depth ≤ 4
Level 2	General	Arbitrary number of triangles in a scene Arbitrary number of objects in a scene
	BBA	Number of bones attachable to an object ≤ 300 Number of bones affecting a vertex ≤ 10

Table 86 — Levels and functionalities constraints

Level	Data Constraints	
Level 1	General	Triangle only Full support of colorIndex, normalIndex, texcoordIndex The vertices of a face shall not define a self-intersecting polygon A given coordIndex is not repeated in a face Color and Texture can be blended.
	Footprint and WSS	"Back Channel" not supported
Level 2	General	Full support of colorIndex, normalIndex, texcoordIndex The vertices of a face shall not define a self-intersecting polygon A given coordIndex is not repeated in a face Color and Texture can be blended.
	Footprint and WSS:	"Back Channel" supported

Table 87 — Levels and player constraints

Level	Player Constraints (Informative Recommendation)
Level 1	Size of Texture memory ≤ 4 M Number of light ≤ 8 Directional Light, PointLight and SpotLight Material: ambientIntensity, diffuseColor, emissiveColor, shininess, specularColor Transparency Flat and Gouraud shading Texture Blending Texture Filtration Perspective correction Total Decoding Time ≤ 10sec